CS150: Database & Datamining

Lecture 29: NoSQL II

Xuming He

Spring 2019

Acknowledgement: Slides are adopted from the Berkeley course CS186 by Joey Gonzalez and Joe Hellerstein, Stanford CS145 by Peter Bailis, IIT Course 236363, Masaryk University PA195 by David Novak.

From RDBMS to NoSQL

- Efficient implementations of table joins and of transactional processing require centralized system.

- NoSQL Databases:
  - Database schema tailored for specific application
    - keep together data pieces that are often accessed together
  - Write operations might be slower but read is fast
  - Weaker consistency guarantees

- => efficiency and horizontal scalability

Data Model

- The model by which the database organizes data
- Each NoSQL DB type has a different data model
  - Key-value, document, column-family, graph
  - The first three are oriented on aggregates

Aggregates

- A data unit with a complex structure
  - Not simply a tuple (a table row) like in RDBMS
- A collection of related objects treated as a unit
  - unit for data manipulation and management of consistency
- Relational model is aggregate-ignorant
  - It is not a bad thing, it is a feature
  - Allows to easily look at the data in different ways
  - Best choice when there is no primary structure for data manipulation

Aggregate example

```
// collection "Customer"
{
  "customerID": 1,
  "name": "Jan Novák",
  "address": {
    "city": "Praha",
    "street": "Krásná 5",
    "ZIP": "111 00"
  }
}

// collection "Invoice"
{
  "invoiceID": 2015003,
  "orderNumber": 11,
  "bankAccount": "64640439/0100",
  "paymentDate": "2015-04-16",
  "address": {
    "city": "Brno",
    "street": "Slunečná 7",
    "ZIP": "602 00"
  }
}
```

```
// collection "Order"
{
  "orderNumber": 11,
  "date": "2015-04-01",
  "customerID": 1,
  "orderItems": [
    {
      "productID": 111,
      "name": "Vysavač ETA E1490",
      "quantity": 1,
      "price": 1300
    },
    {
      "productID": 112,
      "name": "Sáček k ETA E1490",
      "quantity": 10,
      "price": 300
    }
  ]
}
```

NoSQL Databases: Aggregate-oriented

- Many NoSQL stores are aggregate-oriented:
  - There is no general strategy to set aggregate boundaries
  - Aggregates give the database information about which bits of data will be manipulated together
    - What should be stored on the same node
  - Minimize the number of nodes accessed during a search
  - Impact on concurrency control:
    - NoSQL databases typically support atomic manipulation of a single aggregate at a time

Distribution Models: Overview

- Horizontal scalability = scaling out
- Two generic ways of data distribution:
  - Replication – the same data is copied over multiple nodes
    - Master-slave or peer-to-peer
  - Sharding – different data chunks are put on different nodes (data partitioning)

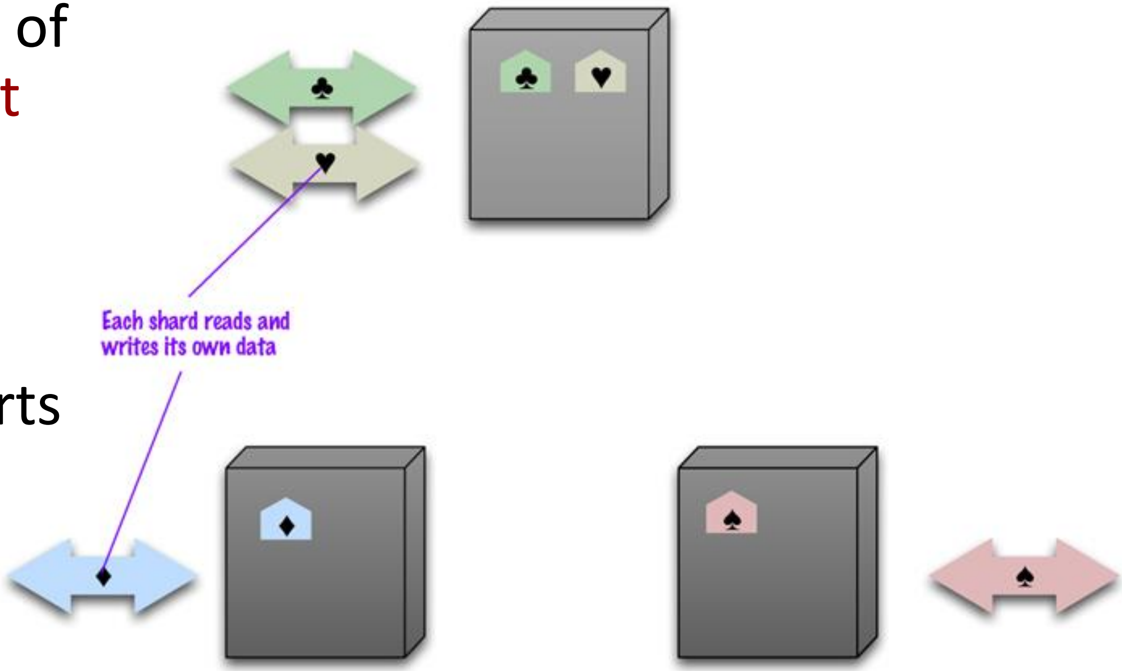
- We can use either or combine them
  - Distribution models = specific ways to do sharding, replication or combination of both

Distribution Model: Single Server

- Running the database on a single machine is always the prefered scenario
  - it spares us a lot of problems
- It can make sense to use a NoSQL database on a single server
  - Other advantages remain: Flexible data model, simplicity
  - Graph databases: If the graph is “almost” complete, it is difficult to distribute it

Sharding (Data Partitioning)

- Placing different parts of the data onto different servers
- Different people are accessing different parts of the dataset



source: Sadalage & Fowler: NoSQL Distilled, 2012

Distribution Models: Sharding (2)

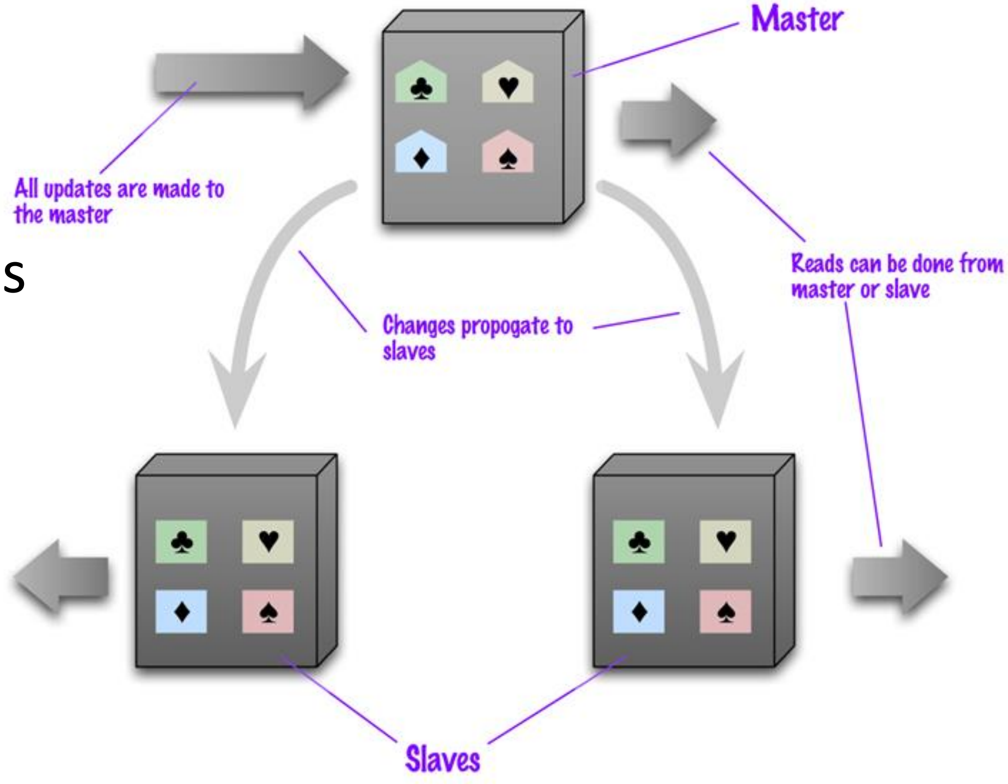
We should try to ensure that

1. Data accessed together is kept together
   - So that user gets all data from a single server
   - Aggregates data model helps to achieve this
2. Arrange the data on the nodes:
   - Based on a physical location (of the data centers)
   - Keep the load balanced (can change in time)

- Many NoSQL databases offer auto-sharding
- A node failure makes shard's data unavailable
  - Sharding is often combined with replication

Master-slave Replication

- We replicate data across multiple nodes
- One node is designated as primary (master), others as secondary (slaves)
- Master is responsible for processing all updates to the data
- Reads from any node



source: Sadalage & Fowler: NoSQL Distilled, 2012

Master-slave Replication (2)

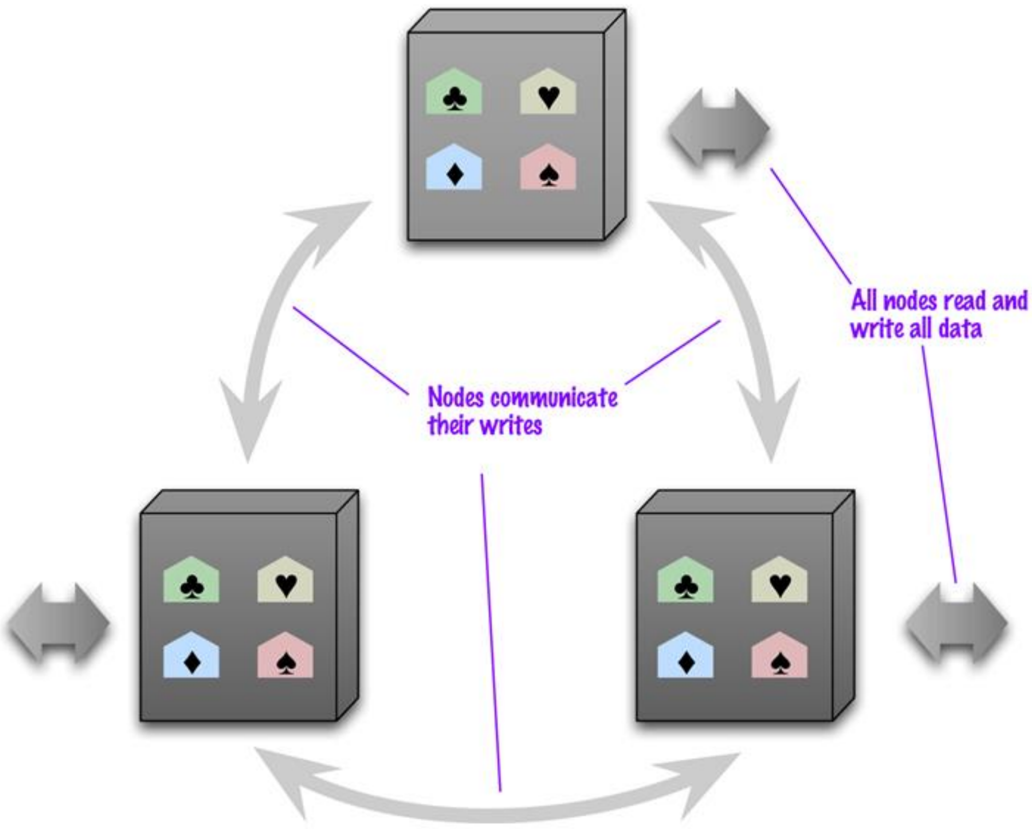
- For scaling a read-intensive application
  - More read requests → more slave nodes
  - The master fails → the slaves can still handle read requests
  - A slave can become a new master quickly (it is a replica)

- Limited by ability of the master to process updates

- Masters are selected manually or automatically
  - User-defined vs. cluster-elected

Peer-to-peer Replication

- No master, all the replicas are equal
- Every node can handle a write and then spreads the update to the others



source: Sadalage & Fowler: NoSQL Distilled, 2012

Peer-to-peer Replication (2)

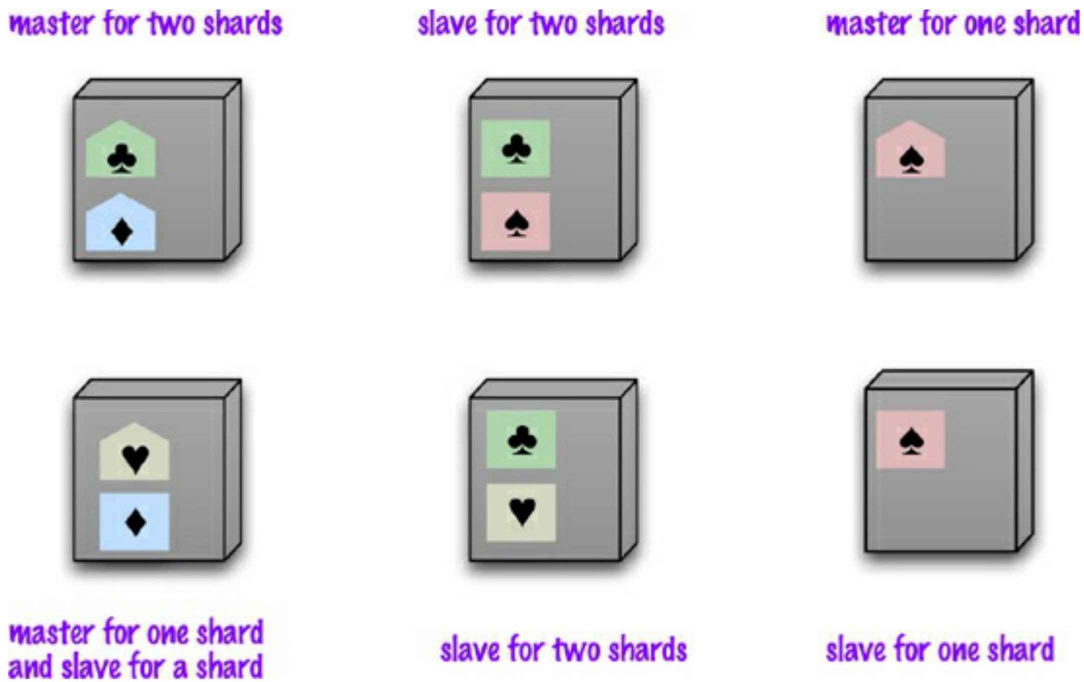
- Problem: consistency
  - Users can write simultaneously at two different nodes

- Solution:
  - When writing, the replicas coordinate to avoid conflict
    - At the cost of network traffic
    - The write operation waits till the coordination process is finished

  - Not all replicas need to agree on the write, just a majority (details below)

Sharding & Replication (1)

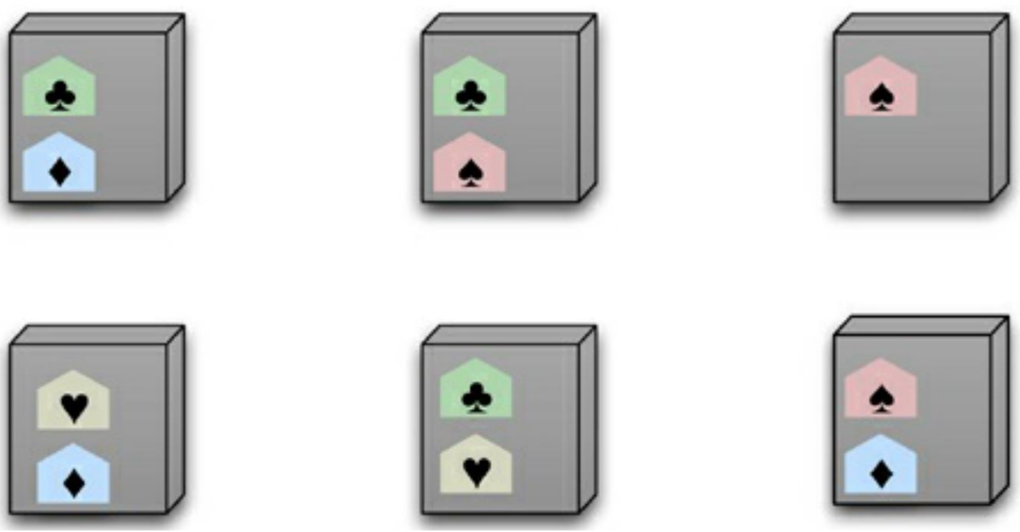
- Sharding and master-slave replication:
  - Each data shard is replicated (via a single master)
  - A node can be a master for some data and a slave for other



source: Sadalage & Fowler: NoSQL Distilled, 2012

Sharding & Replication (2)

- Sharding and peer-to-peer replication:
  - A common strategy for column-family databases
  - A typical default is replication factor of 3
    - each shard is present on three nodes



=> we have to solve consistency issues

(let's first talk more about what consistency means)

source: Sadalage & Fowler: NoSQL Distilled, 2012

Consistency in Databases

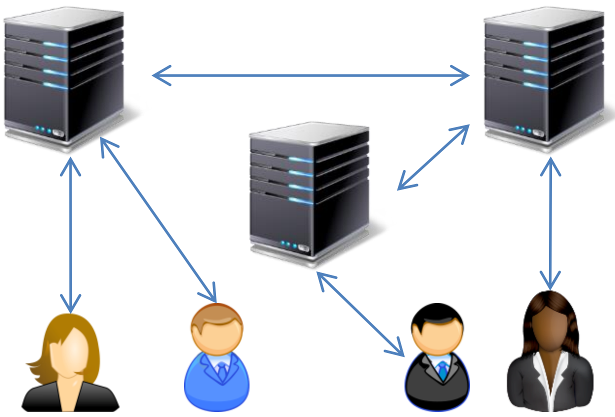
- "Consistency is the lack of contradiction in the DB"
- Centralized RDBMS ensure strong consistency

- Distributed NoSQL databases typically relax consistency (and/or durability)
  - Strong consistency → eventual consistency
  - BASE (basically available, soft state, eventual consistency)
  - CAP theorem
  - tradeoff between consistency and availability

ACID May Be Overly Expensive

- In quite a few modern applications:
  - ACID contrasts with key desiderata: high volume, high availability
  - We can live with some errors, to some extent
  - Or more accurately, we prefer to suffer errors than to be significantly less functional
- *Can this point be made more "formal"?*

Simple Model of a Distributed Service



- Context: distributed service
  - e.g., social network
- Clients make get / set requests
  - e.g., setLike(user,post), getLikes(post)
  - Each client can talk to any server
- Servers return responses
  - e.g., ack, {user_1,....,user_k}
- Failure: the network may occasionally disconnect due to failures (e.g., switch down)
- Desiderata: **C**onsistency, **A**vailability, **P**artition tolerance

CAP Theorem

CAP = Consistency, Availability, Partition Tolerance

Consistency

- After an update, all readers in a distributed system (assuming replication) see the same data
- Example:
  - A single database instance is always consistent
  - If multiple instances exist, the system must handle the writes and/or reads in a special way

CAP Theorem (2)

Availability

- If a node (server) is working, it can read and write data
  - Every request must result in a response

Partition Tolerance

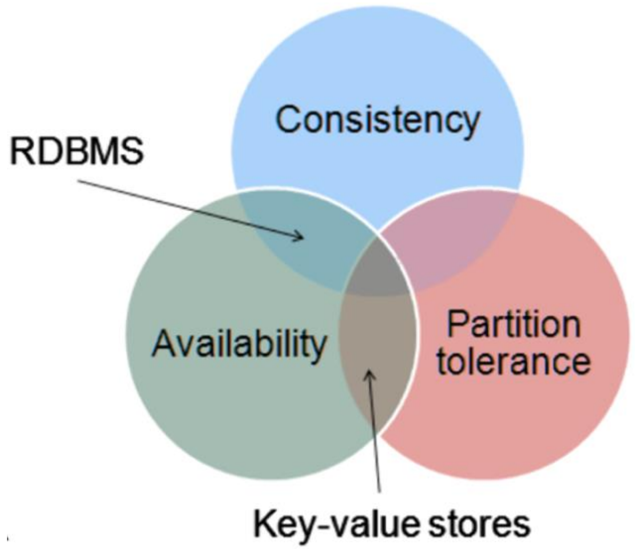
- System continues to operate, even if two sets of servers get isolated
  - A connection failure should not shut the system down

It would be great to have all these three CAP properties!

CAP Theorem: Formulation

- CAP Theorem: A “shared-data” system cannot have all three CAP properties
    - Or: only two of the three CAP properties are possible
        - This is the common version of the theorem
- First formulated in 2000: prof. Eric Brewer
    - PODC Conference Keynote speech
        - www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf
- Proven in 2002: Seth Gilbert & Nancy Lynch
    - SIGACT News 33(2) http://dl.acm.org/citation.cfm?id=564601

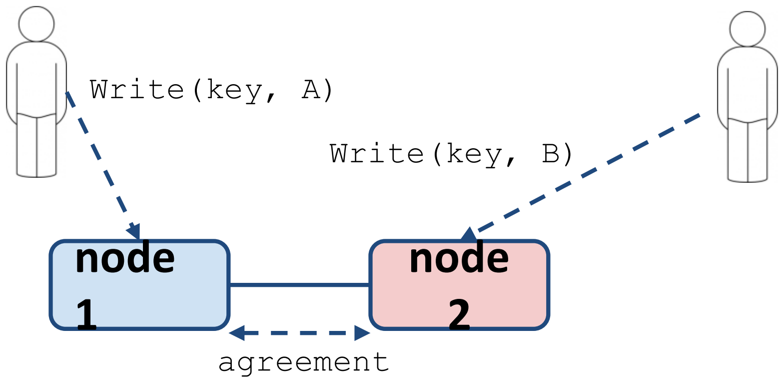
CAP Theorem: Real Application



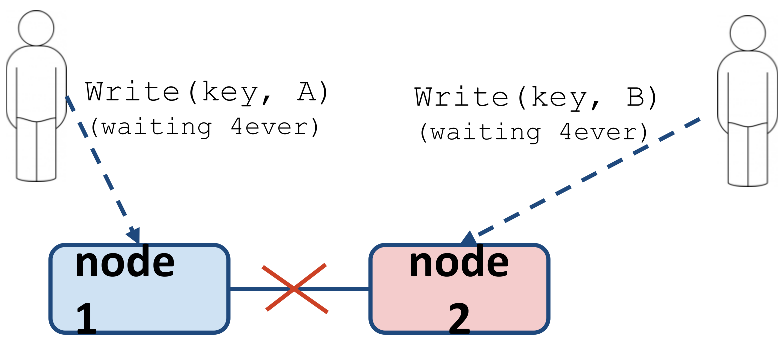
- A single-server system is always CA
  - As well as all ACID systems
- A distributed system practically has to be tolerant of network Partitions (P)
  - because it is difficult to detect all network failures
- So, tradeoff between Consistency and Availability
  - in fact, it is not a binary decision

PC: Partition Tolerance & Consistency

Example: two users, two nodes, two write attempts



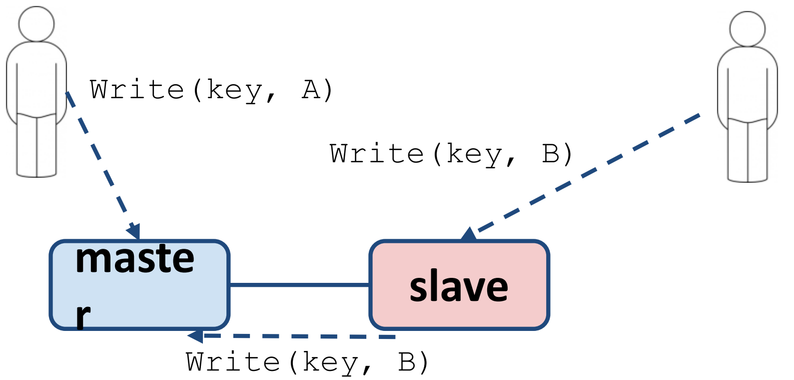
- Strong consistency:
  - Before the write is committed, both nodes have to agree on the order of the writes



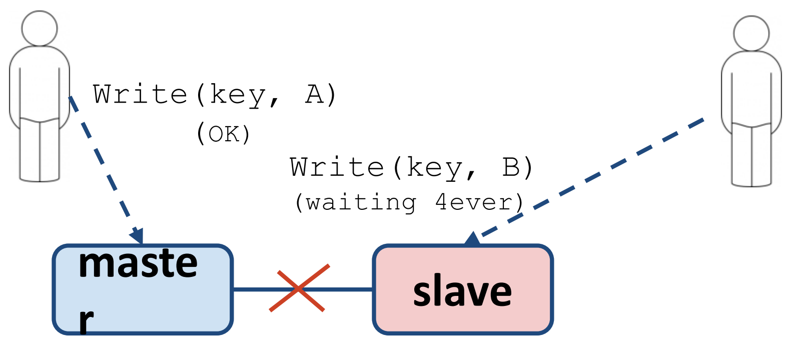
- If the nodes are partitioned, we are losing Availability
  - (but reads are still available)

PC: Partition Tolerance & Consistency (2)

- Adding some availability:
  - Master-slave replication

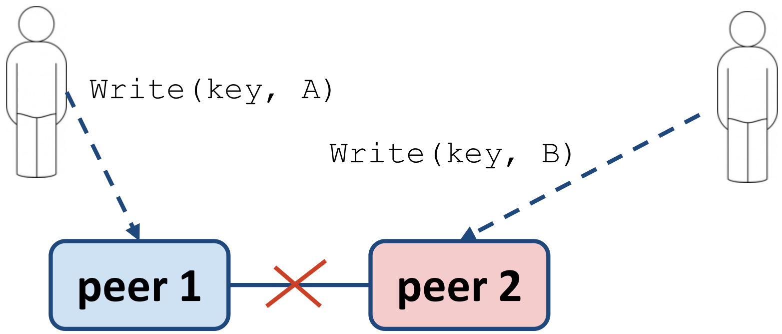


- In case of partitioning, master can commit write
  - Losing some Consistency: Data on slave will be stale for read



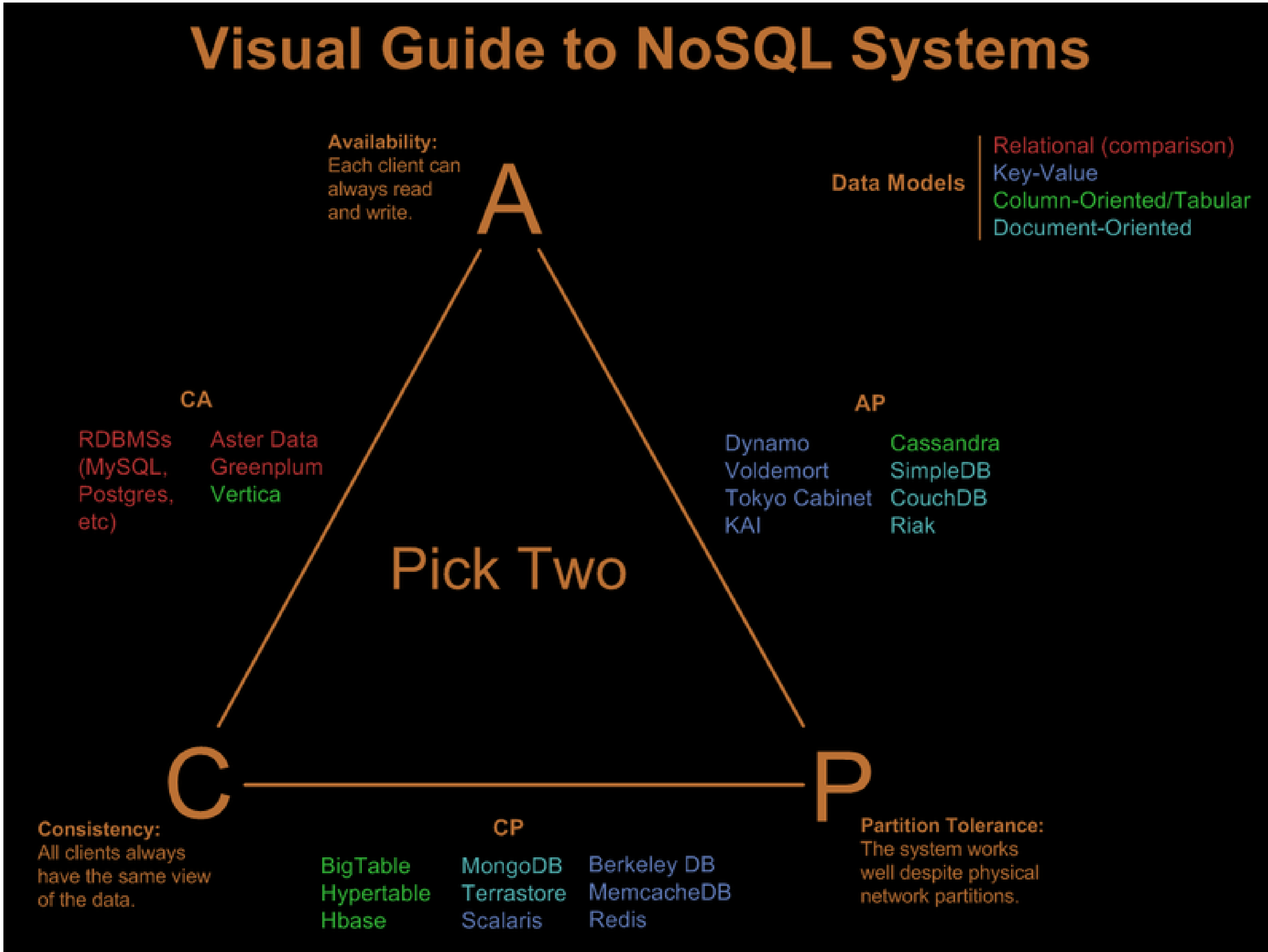
PA: Partition Tolerance & Availability

- Choosing Availability:
  - Peer-to-peer replication
  - Eventual consistency



- In case of Partitioning
  - All requests are answered (full Availability)
  - We risk losing consistency guarantees completely

- But we can do something in the middle: Quorums



2010 visual by Nathan Hurst

http://blog.nahurst.com/visual-guide-to-nosql-systems

The BASE Model

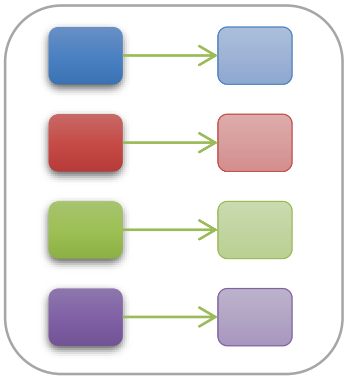
- Applies to distributed systems of type AP
- **B**asic **A**vailability
  - Provide high availability through distribution
- **S**oft state
  - Inconsistency (stale answers) allowed
- **E**ventual consistency
  - If updates stop, then after some time consistency will be achieved
    - Achieved by protocols to propagate updates and verify correctness of propagation (gossip protocols)
- Philosophy: best effort, optimistic, staleness and approximation allowed

Early “Proof of Concepts”

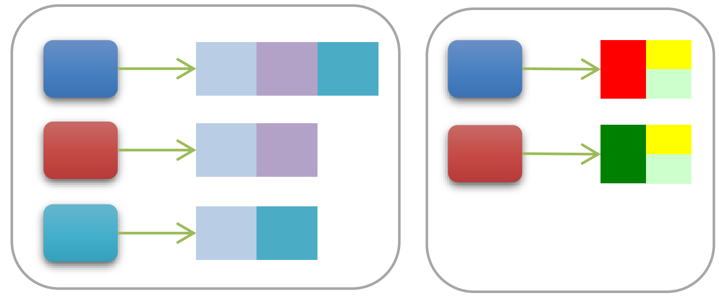
- Memcached: demonstrated that in-memory indexes (DHT) can be highly scalable
- Dynamo: pioneered *eventual consistency* for higher availability and scalability
- BigTable: demonstrated that persistent record storage can be scaled to thousands of nodes

Data Models in NoSQL DBs

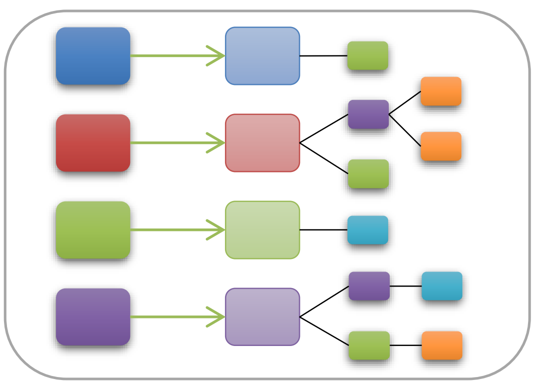
We Will Look at 4 Data Models



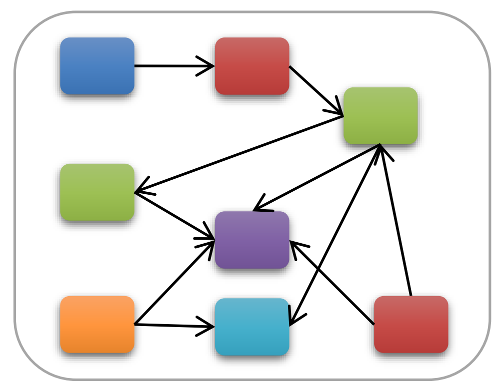
Key/Value Store



Column-Family Store



Document Store



Graph Databases

Key-value Stores: Basics

- A simple hash table (map), primarily used when all accesses to the database are via primary key
  - key-value mapping
- In RDBMS world: A table with two columns:
  - `ID` column (`primary key`)
  - `DATA` column storing the value (unstructured `BLOB`)
- Basic operations:
  - Put a value for a key `put(key, value)`
  - Get the value for the key `value:= get(key)`
  - Delete a key-value `delete(key)`

Querying

- We can query by the key
- To query using some *attribute* of the value is not possible (in general)
  - We need to read the value to test any query condition

- What if we do not know the key?
  - Some systems support additional functionality
    - Using some kind of additional index (e.g. full text)
    - The data must be indexed first
    - Example later: Riak search

Representatives

amazon DynamoDB

riak

levelDB

Infinispan

redis

ORACLE BERKELEY DB 12c

RocksDB

Project Voldemort

MEMCACHED

MapDB

hazelcast

EHCACHE

Ranked list: http://db-engines.com/en/ranking/key-value+store

Selected Challenges & Solutions

Challenge	Selected Techniques
Data partitioning (sharding)	Consistent hashing
Read scalability & reliability	Data replication
Replica management	Version stamps, vector clocks
Detection of a node join/leave/failure	Gossip protocol (no centralized registry of nodes' membership and liveness)
Concurrency, transactions	Two-phase commit protocol, MVCC

Version Stamps

Family of techniques: avoid/detect update conflicts

- Version stamp in general:
  - A field created for each record
  - The stamp changes every time the data record changes

- Basic usage (also in centralized system):
  - A client reads the stamp together with the record
  - When later updating the record, the stamp is sent back together with the new value and checked
  - If the stamp differs from the actual stamp => conflict

Conflict Resolution

- There are three general ways to resolve conflicts
  - (reconcile differences between copies of distributed data)
  - this process is often known as anti-entropy

1. Write repair
   - The correction takes place during a write operation
2. Read repair
   - The correction is done when a read finds an inconsistency
     - Optimistic strategy, read operation is slowed down
3. Asynchronous repair
   - The correction is done as separate operations
   - AKA active "anti-entropy"

Gossip Protocols

A set of distributed protocols

- Each node periodically sends its current info
  - To a randomly-selected peer
  - The peers keep the newer info

In distributed NoSQL databases, gossip is used for

- Spreading information about current state
  - of the entering/leaving/failing nodes
  - asynchronous reconciling of conflicts (anti-entropy)
  - other properties, ...

Redis

- Basically a data structure for strings, numbers, hashes, lists, sets
- Simplistic “transaction” management
  - Queuing of commands as blocks, really
  - Among ACID, only Isolation guaranteed
    - A block of commands that is executed sequentially; no transaction interleaving; no roll back on errors
- In-memory store
  - Persistence by periodical saves to disk
- Comes with
  - A command-line API
  - Clients for different programming languages
    - Perl, PHP, Rubi, Tcl, C, C++, C#, Java, R, …

Example of Redis Commands

key	value

```
get x
>> 10
```

```
hget h y
>> 5
```

```
hkeys p:22
>> name , age
```

```
smembers s
>> 20 , Alma
```

```
scard s
>> 2
```

```
llen l
>> 3
```

```
lrange l 1 2
>> a , b
```

```
lindex l 2
>> b
```

```
lpop l
>> c
```

```
rpop l
>> b
```

Example of Redis Commands

	key	value
(simple value) set x 10	x	10
(hash table) hset h y 5	h	y→5
hset h1 name two hset h1 value 2	h1	name→two value→2
hmset p:22 name Alma age 25	p:22	name→Alma age→25
(set) sadd s 20 sadd s Alma sadd s Alma	s	{20,Alma}
(list) rpush l a rpush l b lpush l c	l	(c,a,b)

```
get x
>> 10
```

```
hget h y
>> 5
```

```
hkeys p:22
>> name , age
```

```
smembers s
>> 20 , Alma
```

```
scard s
>> 2
```

```
llen l
>> 3
```

```
lrange l 1 2
>> a , b
```

```
lindex l 2
>> b
```

```
lpop l
>> c
```

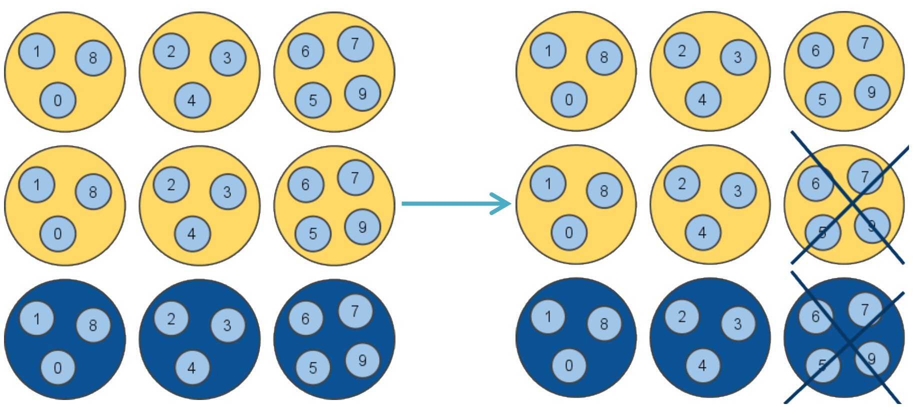
```
rpop l
>> b
```

Additional Notes

- A key can be any <256MB binary string
  - For example, JPEG image
- Some key operations:
  - List all keys: `keys *`
  - Remove all keys: `flushall`
  - Check if a key exists: `exists k`
- You can configure the persistency model
  - `save m k` means save every `m` seconds if at least `k` keys have changed

Redis Cluster

- ➢ Add-on module for managing multi-node applications over Redis
- ➢ Master-slave architecture for sharding + replication
  - Multiple masters holding pairwise disjoint sets of keys, every master has a set of slaves for replication and sharding



http://redis.io/presentation/Redis_Cluster.pdf

K-V Stores: Suitable Use Cases

- Storing Web Session Information
  - Every web session is assigned a unique session_id value
  - Everything about the session can be stored by a single PUT request or retrieved using a single GET
  - Fast, everything is stored in a single object
- User Profiles, Preferences
  - Every user has a unique user_id/user_name + preferences (language, time zone, design, access rights, …)
  - As in the previous case: Fast, single object, single GET/PUT
- Shopping Cart Data
  - Similar to the previous cases

K-V Stores: When Not to Use

- Relationships among Data
  - Relationships between different sets of data
    - Some key-value stores provide link-walking features
- Multi-operation Transactions
  - Saving multiple keys
    - Failure to save any of them → revert or roll back the rest of the operations
- Query by Data
  - Search the keys based on something found in the value part
    - Additional indexes needed (some stores provide them)
- Operations by Key Sets
  - Operations are limited to one key at a time
    - No way to operate upon multiple keys at the same time

K-V Stores: Features & Differences

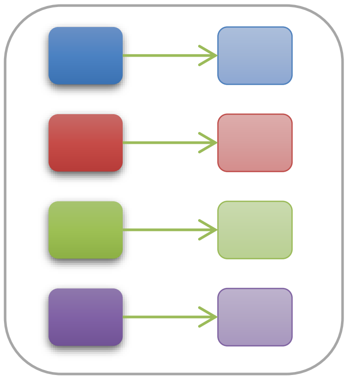
Dozens of key-value stores - how to choose?

1. **Basic** information
   - programming language, license etc.
2. **Internal** Features
   - how are certain principles implemented
   - which influences performance/security/reliability/etc.
3. **Advanced** (User-visible) Features
   - what “advanced” features does the store provide
     - besides store/get/delete operations

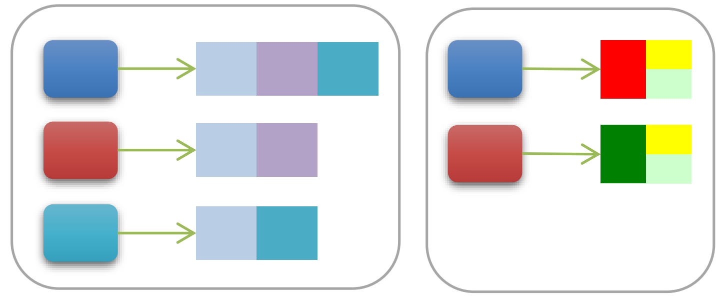
Key-Value Stores Summary

- Essentially, big distributed hash maps
- Origin attributed to Dynamo – Amazon's DB for world-scale catalog/cart collections
  - But Berkeley DB has been here for >20 years
- Store pairs ⟨key,opaque-value⟩
  - Opaque means that DB does not associate any structure/semantics with the value; *oblivious* to values
  - This may mean more work for the user: retrieving a large value and parsing to extract an item of interest
- Sharding via partitioning of the key space
  - Hashing, gossip and remapping protocols for load balancing and fault tolerance

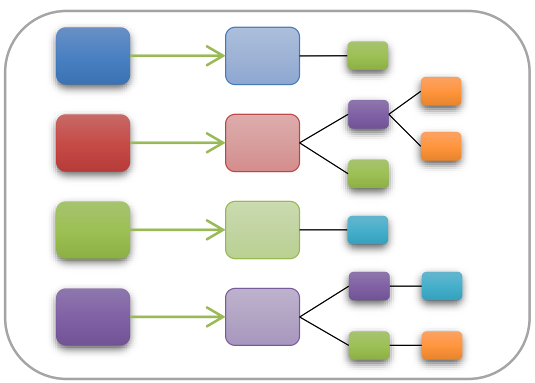
We Will Look at 4 Data Models



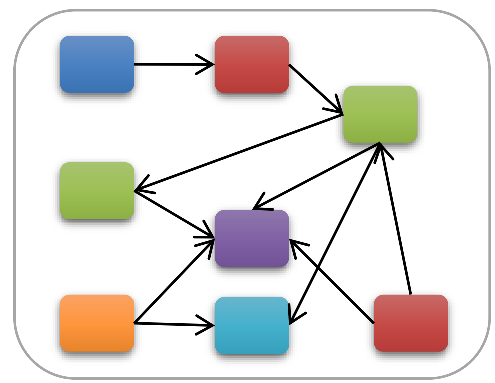
Key/Value Store



Column-Family Store



Document Store



Graph Databases

Data Formats

- Binary Data (KV store)
  - often, we want to store objects (class instances)
  - objects can be binary serialized (marshalled)
    - and kept in a key-value store
  - there are several popular serialization formats
    - Protocol Buffers, Apache Thrift
- Structured Text Data
  - JSON, BSON (Binary JSON)
    - JSON is currently number one data format used on the Web
  - XML: eXtensible Markup Language
  - RDF: Resource Description Framework

JSON: Basic Information

- Text-based open standard for data interchange
  - Serializing and transmitting structured data
- JSON = JavaScript Object Notation
  - Originally specified by Douglas Crockford in 2001
  - Derived from JavaScript scripting language
  - Uses conventions of the C-family of languages
- Filename: *.json
- Internet media (MIME) type: application/json
- Language independent

http://www.json.org

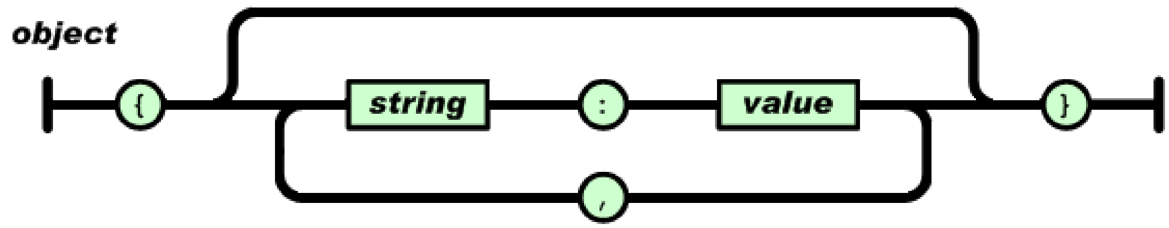
JSON:Example

```json
{
  "conferences":
  [
    {
      "name": "XML Prague 2015",
      "start": "2015-02-13",
      "end": "2015-02-15",
      "web": "http://xmlprague.cz/",
      "price": 120,
      "currency": "EUR",
      "topics": ["XML", "XSLT", "XQuery", "Big Data"],
      "venue": {
        "name": "VŠE Praha",
        "location": {
          "lat": 50.084291,
          "lon": 14.441185
        }
      }
    },
    {
      "name": "DATAKON 2014",
      "start": "2014-09-25",
      "end": "2014-09-29",
      "web": "http://www.datakon.cz/",
      "price": 290,
      "currency": "EUR",
      "topics": ["Big Data", "Linked Data", "Open Data"]
    }
  ]
}
```

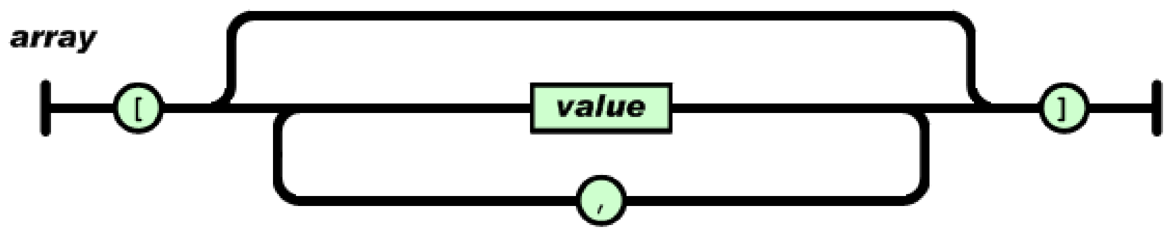
source: I. Holubová, J. Kosek, K. Minařík, D. Novák. Big Data a NoSQL databáze. Praha: Grada Publishing, 2015.

JSON: Data Types (1)

- object – an unordered set of name+value pairs
  - these pairs are called properties (members) of an object
  - syntax: { name: value, name: value, name: value, ...}

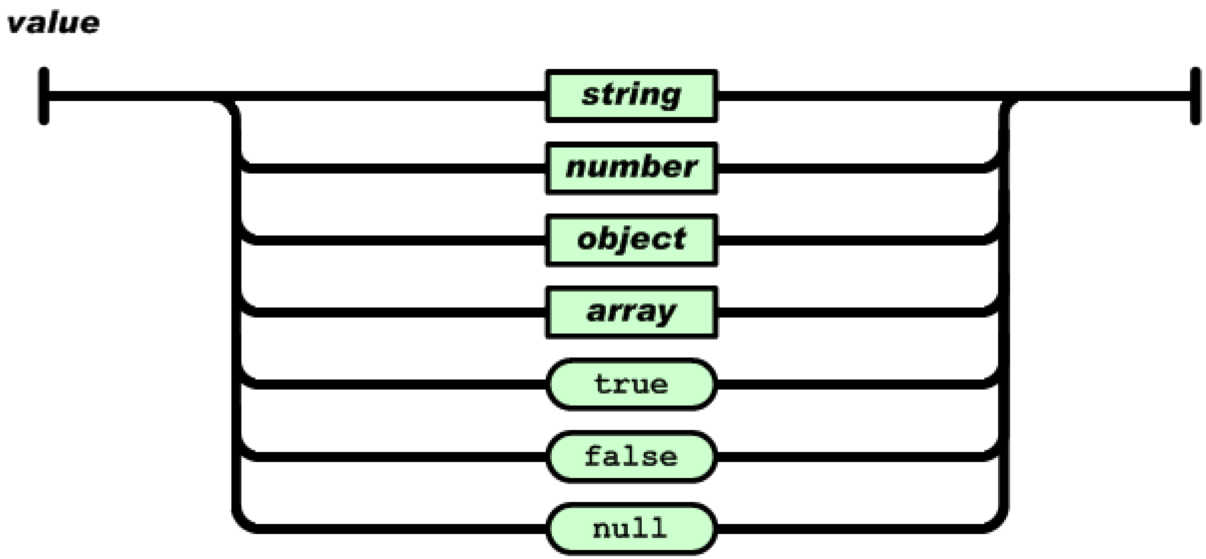


- array – an ordered collection of values (elements)
  - syntax: [comma-separated values]



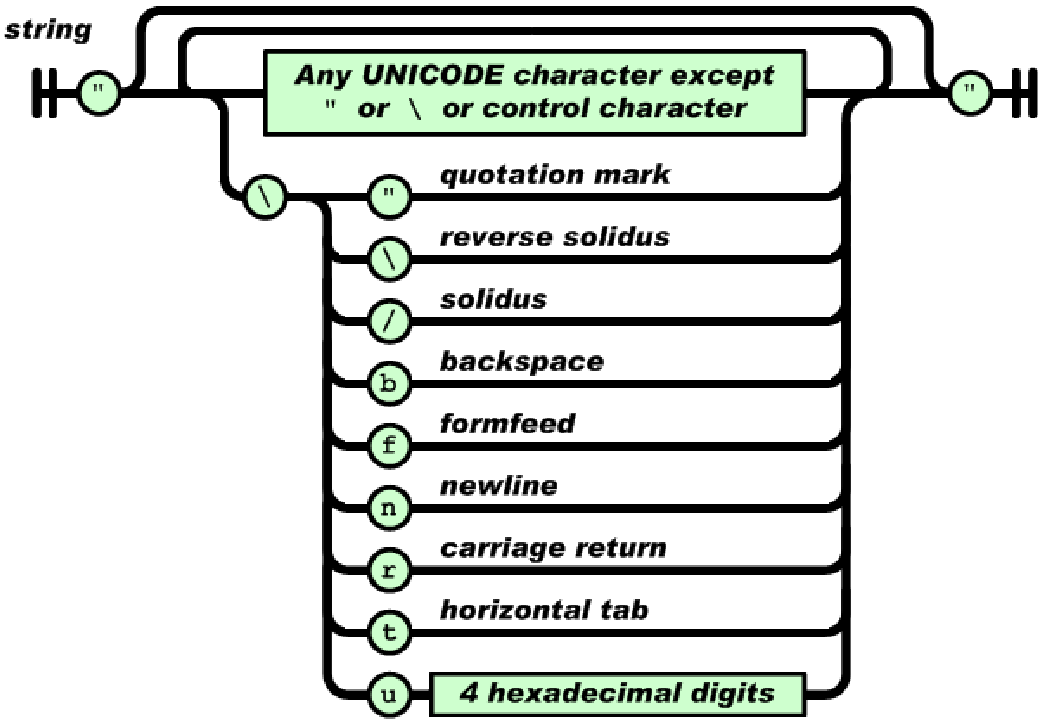
JSON: Data Types (2)

- value – string in double quotes / number / true or false (i.e., Boolean) / null / object / array



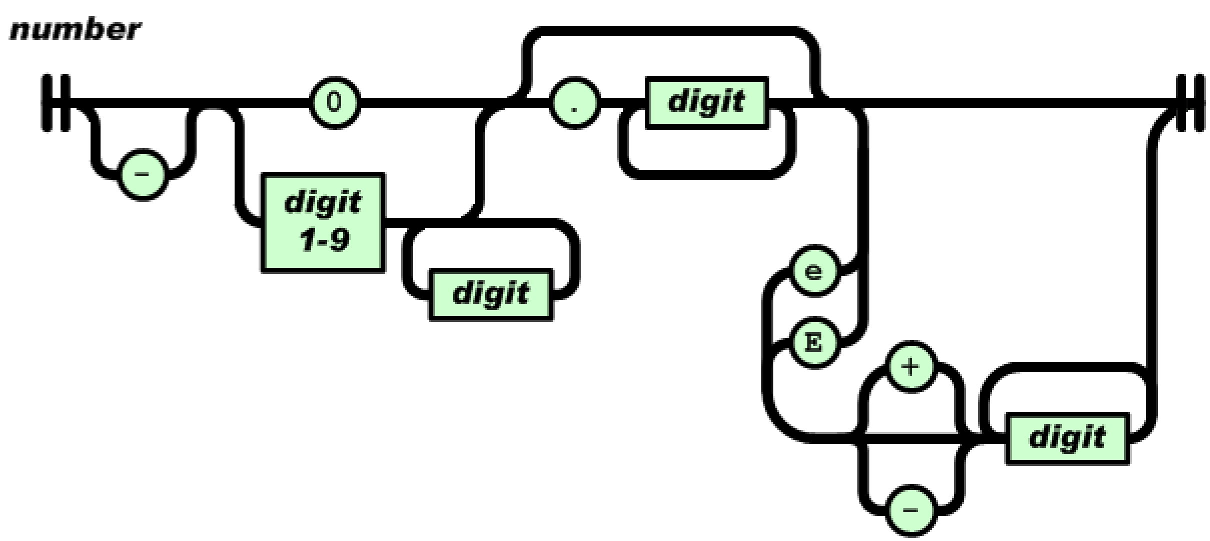
JSON: Data Types (3)

- string – sequence of zero or more Unicode characters, wrapped in double quotes
  - Backslash escaping



JSON: Data Types (4)

- number – like a C or Java number
  - Integer or float
  - Octal and hexadecimal formats are not used



JSON Properties

- There is no way to write comments in JSON
  - Originally, there was but it was removed for security
- No way to specify precision/size of numbers
  - It depends on the parser and the programming language
- There exists a standard “JSON Schema”
  - A way to specify the schema of the data
  - Field names, field types, required/optional fields, etc.
  - JSON Schema is written in JSON, of course
    - see example below

JSON Schema: Example

```
{
  "$schema": "http://json-schema.org/schema#",
  "type": "object",
  "properties": {
    "conferences": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": { "type": "string" },
          "start": { "type": "string", "format": "date" }
          "end": { "type": "string", "format": "date" },
          "web": { "type": "string" },
          "price": { "type": "number" },
          "currency": { "type" : "string",
                  "enum": ["CZK", "USD", "EUR", "GBP"] },
          "topics": {
            "type": "array",
            "items": {
              "type": "string"
            }
          },
          "venue": {
            "type": "object",
            "properties": {
              "name": { "type": "string" },
              "location": {
                "type": "object",
                "properties": {
                  "lat": { "type": "number" },
                  "lon": { "type": "number" }
                }
              }
            },
            "required": ["name"]
          }
        },
        "required": ["name", "start", "end",
                     "web", "price", "topics"]
      }
    }
  }
}
```

source: I. Holubová, J. Kosek, K. Minařík, D. Novák. Big Data a NoSQL databáze. Praha: Grada Publishing, 2015.

Document with JSON Schema

```
{
  "conferences":
  [
    {
      "name": "XML Prague 2015",
      "start": "2015-02-13",
      "end": "2015-02-15",
      "web": "http://xmlprague.cz/",
      "price": 120,
      "currency": "EUR",
      "topics": ["XML", "XSLT", "XQuery", "Big Data"],
      "venue": {
        "name": "VŠE Praha",
        "location": {
          "lat": 50.084291,
          "lon": 14.441185
        }
      }
    },
    {
      "name": "DATAKON 2014",
      "start": "2014-09-25",
      "end": "2014-09-29",
      "web": "http://www.datakon.cz/",
      "price": 290,
      "currency": "EUR",
      "topics": ["Big Data", "Linked Data", "Open Data"]
    }
  ]
}
```

source: I. Holubová, J. Kosek, K. Minařík, D. Novák. Big Data a NoSQL databáze. Praha: Grada Publishing, 2015.

XML: Basic Information

- XML: eXtensible Markup Language
  - W3C standard (since 1996)
- both human and machine readable
- example:

```
<?xml version="1.0"?>
<quiz>
 <qanda seq="1">
  <question>
   Who was the forty-second
   president of the U.S.A.?
  </question>
  <answer>
   William Jefferson Clinton
  </answer>
 </qanda>
 <!-- Note: We need to add
  more questions later.-->
</quiz>
```

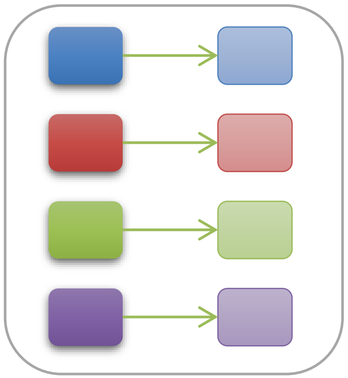
source: http://en.wikipedia.org/wiki/XML

XML: Features and Comparison

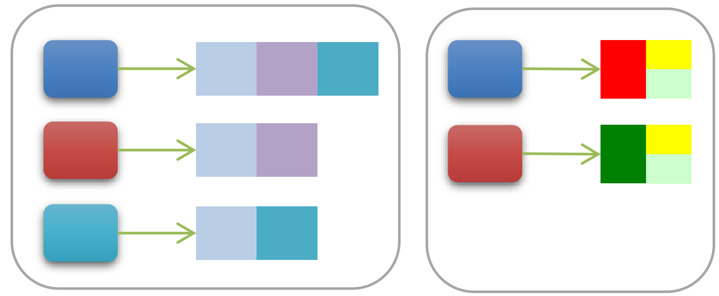
- Standard ways to specify XML document schema:
  - DTD, XML Schema, etc.
  - concept of Namespaces; XML editors (for given schema)
- Technologies for parsing: DOM, SAX
- Many associated technologies:
  - XPath, XQuery, XSLT (transformation)

- XML is great for configurations, meta-data, etc.
- XML databases are mature, not considered NoSQL
- Currently, JSON format rules:
  - compact, easier to write, has all features typically needed

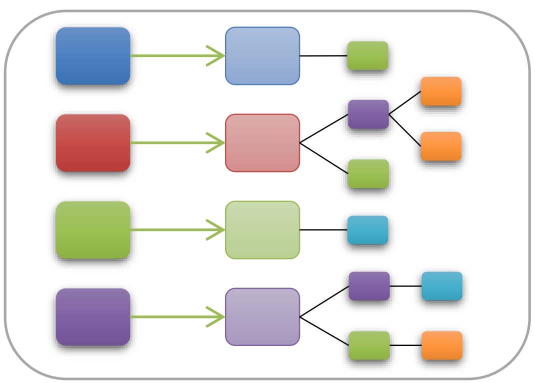
To be continued…



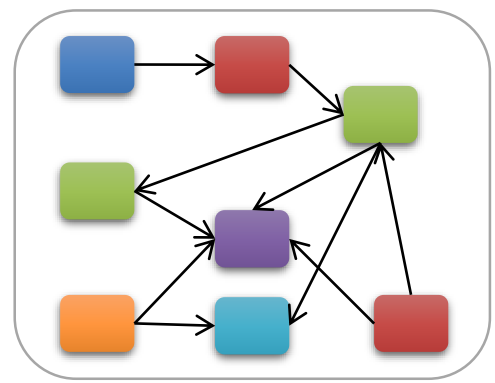
Key/Value Store



Column-Family Store



Document Store



Graph Databases