Lecture 4 - Introduction NOSQL, and MongoDB

Instructor: Suresh Melvin Sigera
Email: suresh.sigera@cuny.csi.edu

Agenda

- What is a schema-less aka NoSQL data model?
- CAP theorem
- What is JSON?
- What is MongoDB?
- QA

What is a schema-less data model?

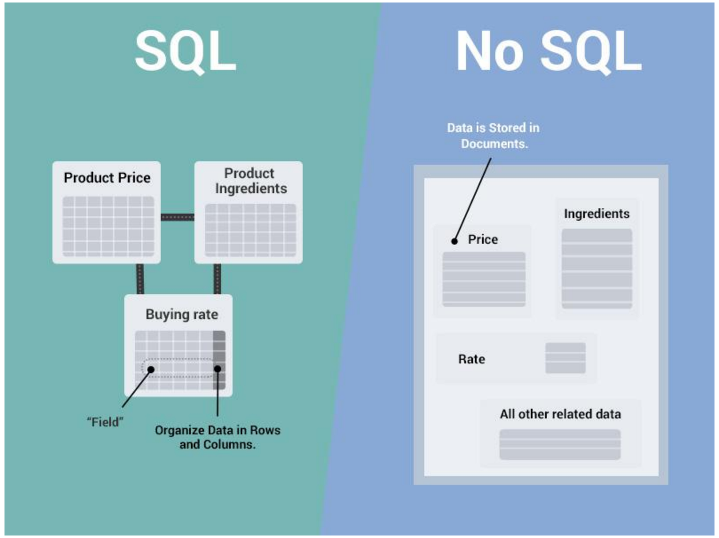
In a relational databases

- You can't add a record which does not fit the schema
- You need to add NULLs to unused items in a row
- We should consider the data type prior to the database design, i.e you can't add a string to an integer field
- You can't add multiple items in a field, you should create another table: primary-key, foreign key, joins and normalization ...!!!

What is a schema-less data model?

In NoSQL databases

- There is no schema to consider
- There is no unused cell
- There is no datatype (implicit)
- Most of the considerations are done in the application layer
- We gather all items as in an aggregate (document)



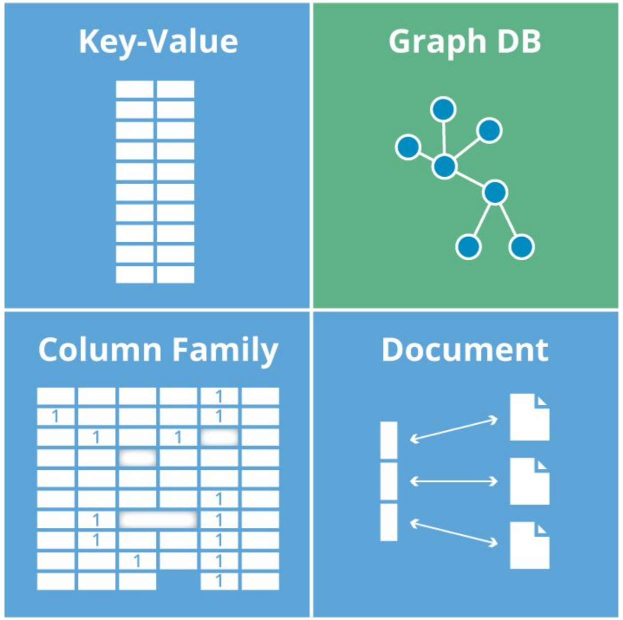
NoSQL databases

What does NoSQL actually mean?

- NoSQL movement = The whole point of **seeking alternatives is that you need to solve a problem that rela onal databases are a bad fit for**
- NoSQL databases = Next genera on databases mostly addressing some of the points: being non-relational, distributed, open-source and horizontally scalable. The original intention on has been modern web-scale databases. Often more characteristics apply as: schema-free, easy replica on support, simple API, eventually consistent, a huge data amount, and more

Types of NoSQL

- Key-value
- Graph database
- **Document oriented**
- Column family



Benefits of NoSQL

- Scaling
    - **Relational databases**
        - **Traditional approach - scaling up, buying bigger servers as database load increases**
    - NoSQL
        - Scale out - distribute data across multiple hosts seamlessly
- Volume AKA Big Data
    - **RDBMS**
        - **Capacity and constraints of data volumes at its limits**
    - NoSQL
        - Huge increase in data

Benefits of NoSQL

- Administrators
    - **Relational databases**
        - **Require highly trained expert to monitor DB**
    - NoSQL
        - Require less management, automatic repair and simpler data models
- Flexibility
    - **Relational databases**
        - **Change management to schema for RDMS have to be carefully managed**
    - NoSQL
        - Databases more relaxed in structure of data
        - Database schema changes do not have to be managed as one complicated change unit
        - Application already written to address an amorphous schema

Benefits of NoSQL

- Economics
    - **Relational databases**
        - **Rely on expensive proprietary servers to manage data**
    - NoSQL
        - Clusters of cheap commodity servers to manage the data and transaction volumes
        - Cost per gigabyte or transaction/second for NoSQL can be lower than the cost for a RDBMS
- Relaxed consistency
    - **Relational databases**
        - **Strong consistency (ACID properties and transactions)**
    - NoSQL
        - Eventual consistency only (BASE properties)
            - I.e. we have to make trade-offs because of the data distribution

The end of relational databases?

- Certainly no
  - RDBMS is a great tool for solving ACID problems
    - When data validity is super important
    - When you need to support dynamic queries
  - NoSQL is great tool for solving data availability problems
    - When it's more important to have fast data than right data; but still it can prevent from application development side
    - When you need to scale bases on the changing requirements
  - Pick the right tool for the job

NoSQL Databases: principles

- Different aspects of data distribution
  - **Scaling**
    - Vertical vs. horizontal
  - Distribution models
    - Sharding
    - Replication: master-slave vs. peer-to-peer architectures
  - CAP properties
    - **Consistency, availability and partition tolerance**
    - ACID vs. **BASE guarantees**

Scalability

What is scalability?

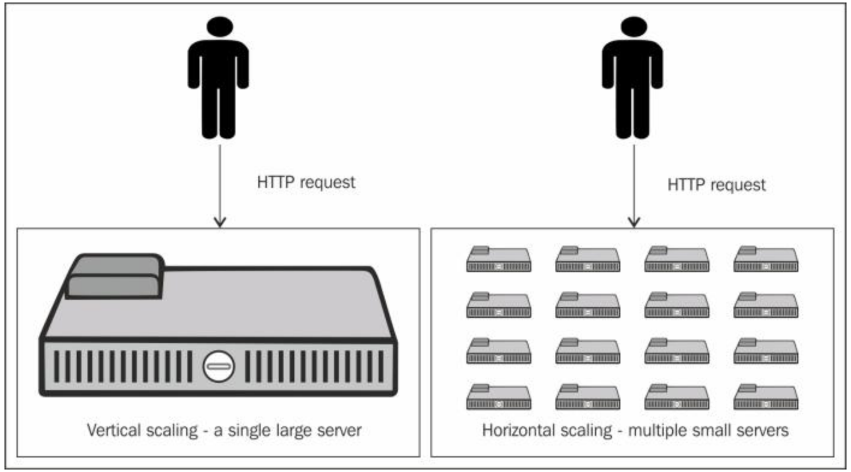
- Capability of a system to handle growing amounts of data and/or queries without losing performance, or its potential to be enlarged in order to accommodate such a growth
- Two general approaches
  - Vertical scaling
  - Horizontal scaling

Scalability

Vertical scaling (scaling up/down)

- Adding resources to a single node in a system
  - E.g. increasing the number of CPUs, extending system memory,
  - using larger disk arrays, …
  - I.e. larger and more powerful machines are involved
- Traditional choice
  - In favor of strong consistency
  - Easy to implement and deploy
  - No issues caused by data distribution
  - No expensive JOINS

Works well in many cases but …


HTTP request
HTTP request
Vertical scaling - a single large server
Horizontal scaling - multiple small servers

Vertical scalability: drawbacks

Performance limits

- Even the most powerful machine has a limit
    - Moreover, everything works well… unless we start approaching such limits
- Higher costs
    - The cost of expansion increases exponentially
    - In particular, it is higher than the sum of costs of equivalent commodity hardware
- Proactive provisioning
    - New projects / applications might evolve rapidly
    - Upfront budget is needed when deploying new machines
    - And so flexibility is seriously suppressed

Vertical scalability: drawbacks

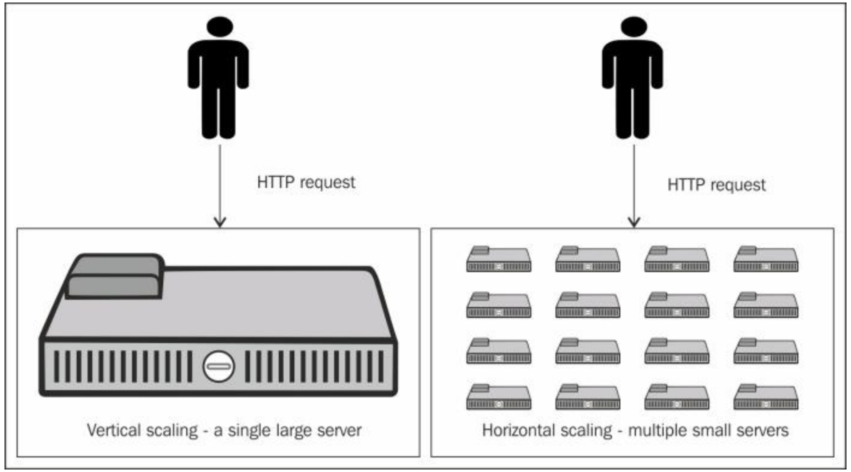
Vendor lock-in

- There are only a few manufacturers of large machines
- Customer is made dependent on a single vendor
  - Their products, services, but also implementation details, proprietary formats, interfaces, …
- I.e. it is difficult or impossible to switch to another vendor
  - The cost of expansion increases exponentially
  - In particular, it is higher than the sum of costs of equivalent commodity hardware
- Deployment downtime
  - Inevitable downtime is often required when scaling up

Horizontal scalability: fallacies

False assumptions

- Network is reliable
- Latency is zero
- Bandwidth is infinite
- Network is secure
- Topology does not change
- There is one administrator
- Transport cost is zero
- Network is homogeneous


HTTP request
HTTP request
Vertical scaling - a single large server
Horizontal scaling - multiple small servers

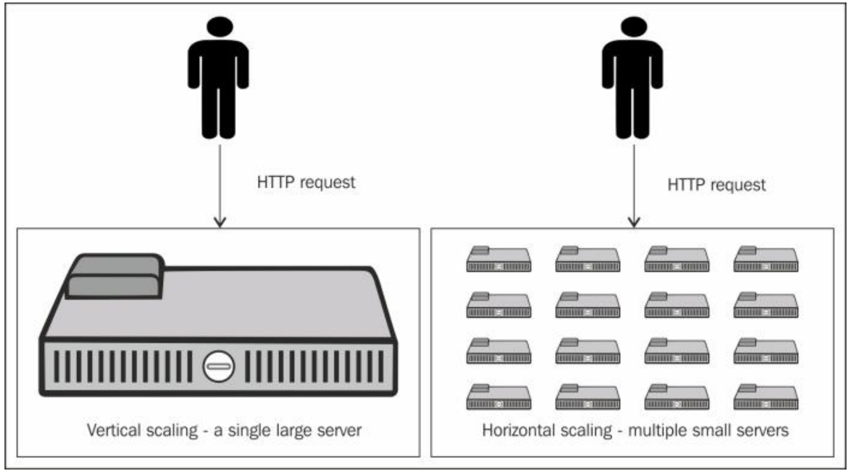
Horizontal scalability: consequences

Significantly increases complexity

- Complexity of management, programming model, …

Introduces new issues and problems

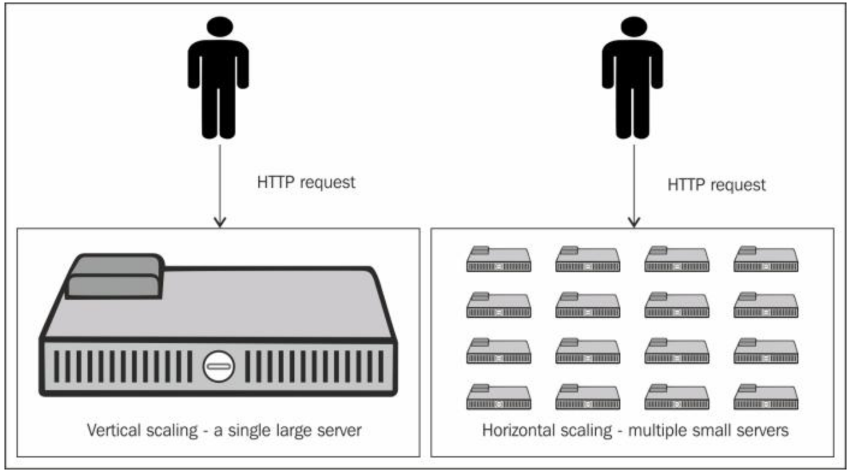
- Synchronization of nodes
- Data distribution
- Data consistency
- Recovery from failures


HTTP request
HTTP request
Vertical scaling - a single large server
Horizontal scaling - multiple small servers

Horizontal scalability: conclusion

A standalone node still might better option in certain cases.

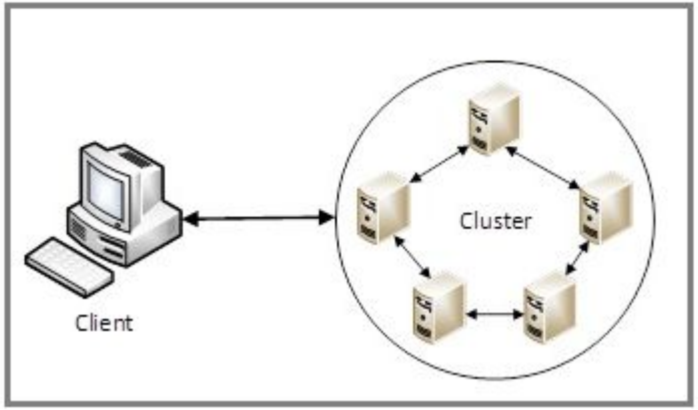
- E.g. for graph databases - Simply because it is difficult to split and distribute graphs
- In other words
  - **It can make sense to run even a NoSQL database system on a single node**
  - No distribution at all is the most preferred / simple scenario
- But in general, horizontal scaling really opens new possibilities


HTTP request
HTTP request
Vertical scaling - a single large server
Horizontal scaling - multiple small servers

Horizontal scalability: architecture

What is a cluster?

- =**a collection of mutually interconnected commodity nodes**
- Based on the **shared-nothing architecture**
  - Nodes do not share their CPUs, memory, hard drives, Each node runs its own operating system instance
  - **Nodes send messages to interact with each other**
- Nodes of a cluster can be heterogeneous
- Data, queries, computation, workload, … this is all **distributed among the nodes** within a cluster



Distribution models

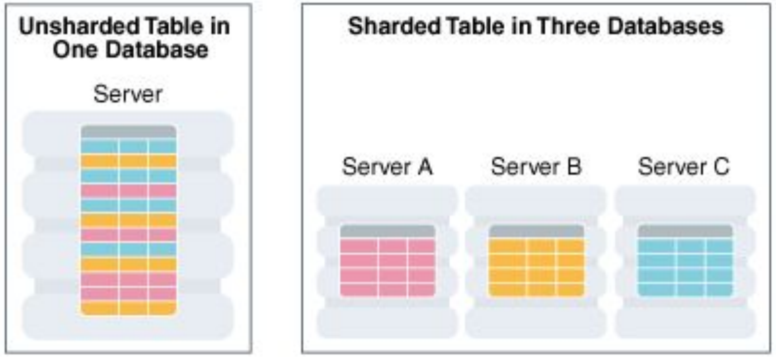
Generic techniques of **data distribution**

- Sharding
  - **Different data on different nodes**
  - Motivation: increasing volume of data, increasing performance
- Replication
  - Copies of the **same data on different nodes**
  - Motivation: increasing performance, increasing fault tolerance
- Both the techniques are mutually orthogonal
  - I.e. we can use either of them, or combine them both
- Distribution model
  - = specific way how sharding and replication is implemented NoSQL systems often offer automatic sharding and replication

Sharding

Sharding (horizontal partitioning)

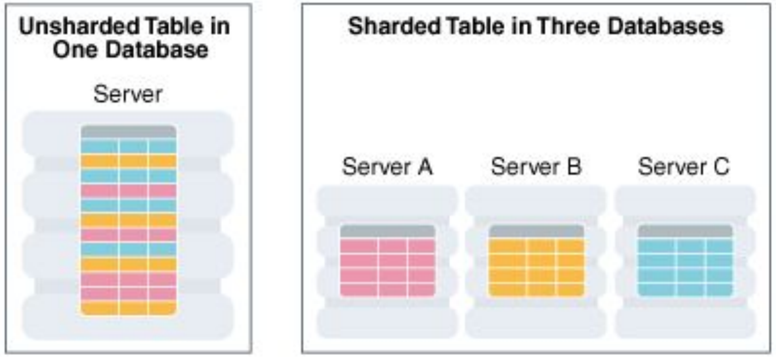
- Placement of different data on different nodes
    - **What different data means? Different aggregates**
        - E.g. key-value pairs, documents, ...
- Related pieces of data that are accessed together should also be kept together
    - Specifically, operations involving data on multiple shards should be avoided


Unsharded Table in One Database
Server
Sharded Table in Three Databases
Server A
Server B
Server C

Sharding

Objectives

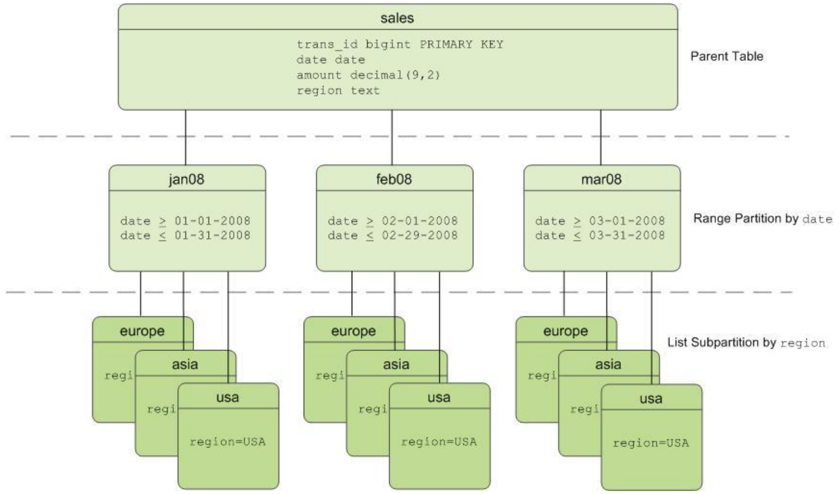
- Uniformly distributed data (volume of data)
- Balanced workload (read and write requests)
- Respecting physical locations
  - E.g. different data centers for users around the world
- Unfortunately, these objectives...
  - may mutually contradict each other
  - may change in time


Unsharded Table in One Database
Server
Sharded Table in Three Databases
Server A
Server B
Server C

Sharding

How to actually determine shards for aggregates?

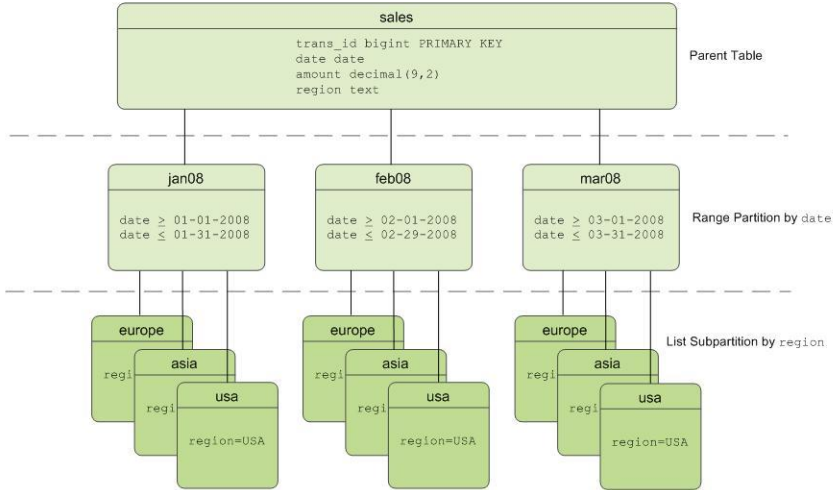
- We not only need to be able to place new data when handling write requests, but also find the data in case of read requests
  - I.e. when a given search criterion is provided (e.g. key, id, ...), we must be able to determine the corresponding shard
    - So that the requested data can be accessed and returned, or failure can be correctly detected when the data is missing



Sharding

Sharding strategies

- Based on mapping structures
  - Placing of data on shards in a random fashion (e.g. round-robin)
  - Mapping of individual aggregates to particular shards must be maintained (this is not a suitable solution)
- Based on general rules: hash partitioning, and range partitioning



Replication

Replication

- **Placement of multiple copies – replicas – of the same data on different nodes**
- **Replica on factor** = the number of copies

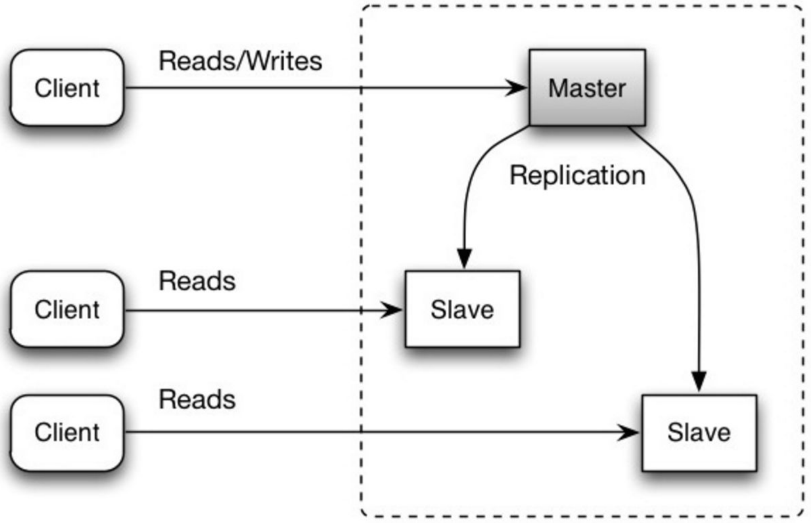
Two approaches

- Master-slave architecture
- Peer-to-peer architecture

Replication - master-slave architecture

Architecture

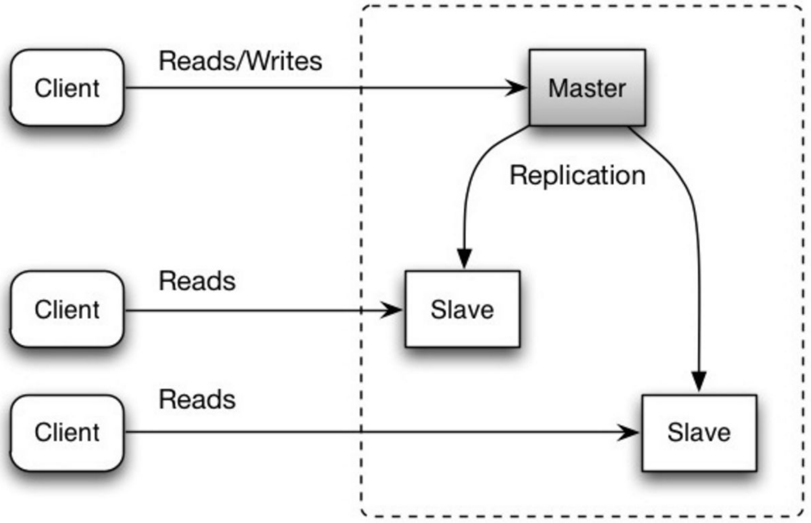
- One node is primary (master), all the other secondary (slave)
- Master node bears all the management responsibility
- All the nodes contain identical data



Replication - master-slave architecture

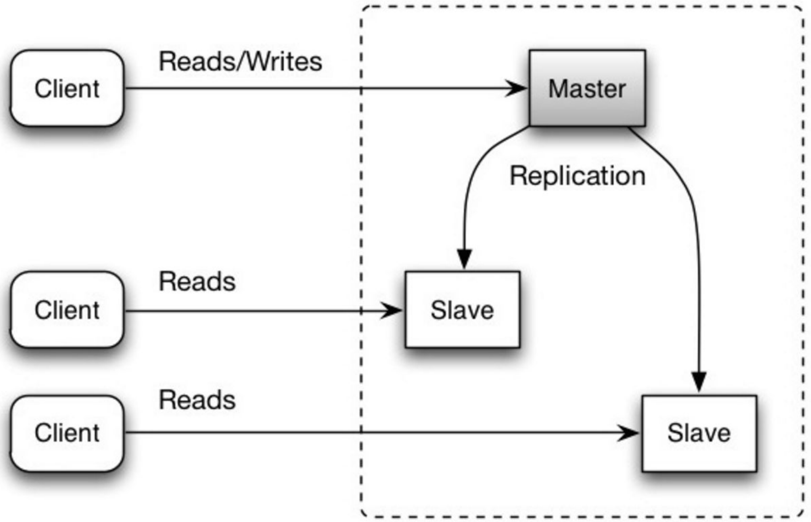
Read requests can be handled by both the master or slaves

- Suitable for read-intensive applications
  - More read requests to deal with → **more slaves to deploy**
  - When the master fails, read operations can still be handled



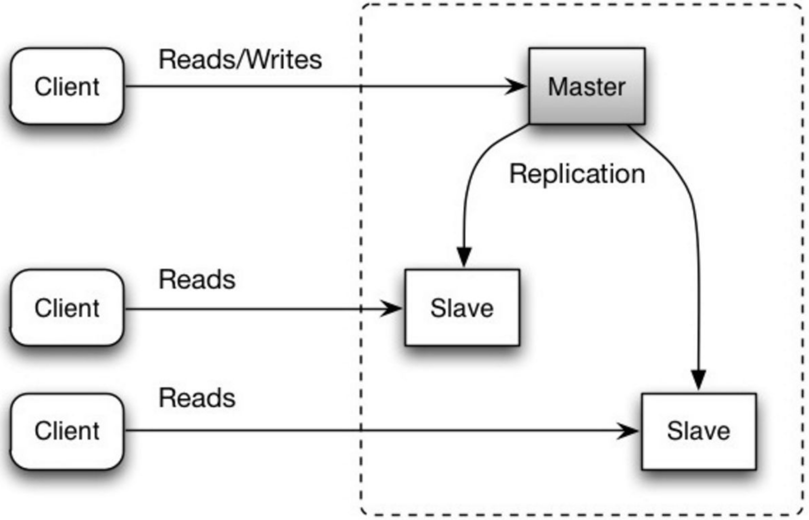
Replication - master-slave architecture

- Write requests can only be handled by the master
- Newly written replicas are propagated to all the slaves
- Consistency issue
  - Luckily enough, at most one write request is handled at a time
  - But the propagation still takes some time during which obsolete reads might happen
  - Hence certain synchronization on is required to avoid conflicts



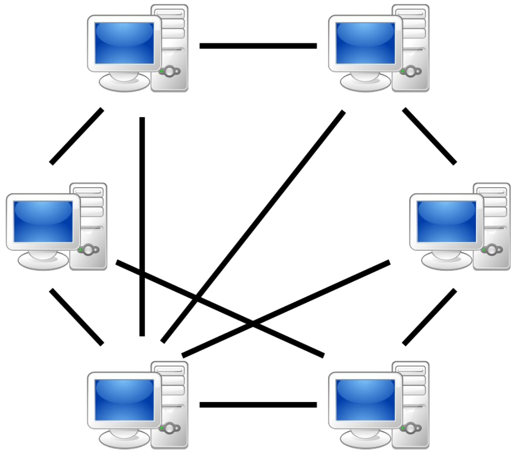
Replication - master-slave architecture

- In case of **master failure**, a new one needs to be appointed
    - Manually (user-defined) or automatically (cluster-elected)
    - Since the nodes are identical, appointment can be fast
- Master might therefore represent a **bottleneck**
    - (because of the performance or failures)



Replication - peer-to-peer architecture

- Architecture
  - **All the nodes have equal roles and responsibilities**
  - All the nodes contain **identical data** once again
- Both read and write requests can be handled by any node
  - No bottleneck, no single point of failure
  - Both the operations scale well
  - More requests to deal with → more nodes to deploy



The CAP theorem

The CAP theorem, also known as Brewer's theorem, states that **it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees**:

- Consistency (**all nodes** see the same data at the same time)
- Availability (a guarantee that **every request receives a response about whether it was successful or failed**)
- Partition tolerance (the **system continues to operate** despite arbitrary message loss or failure of part of the system)

According to the theorem, a distributed system cannot satisfy all three of these guarantees at the same time.

JSON = JavaScript Object Notation

- Open standard for data interchange; It is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate.
- Design goals
  - Simplicity: text-based, easy to read and write
  - Universality: object and array data structures – Supported by majority of modern programming languages
- Derived from JavaScript (but language independent)
- Started in 2002
- File extension: *.json
- Content type: application/json
- http://www.json.org/

```
{
    "title": "Thor",
    "year": 2018,
    "actors": [{
            "firstname": "Chris",
            "lastname": "Hemsworth"
        },
        {
            "firstname": "Tom",
            "lastname": "Hiddleston"
        }
    ],
    "director": {
        "firstname": "Taika",
        "lastname": "Waititi"
    }
}
```

JSON - data structure

Object

- Unordered collection of name-value pairs (properties)
    - Correspond to structures such as objects, records, structs, dictionaries, hash tables, keyed lists, associative arrays, …

Example

- `{ "name" : "Tom Cruise", "dob" : 1692 }`
- `{}`

JSON - data structure

Array

- Ordered collection of values
  - Correspond to structures such as arrays, vectors, lists, sequences ...
  - Values can be of different types, duplicate values are allowed

Example

- `[2, 7, 7, 5]`
- `["Tom Cruise", 1962, 5.7]`
- `[]`

JSON - data structure

Value

- Unicode string
  - Enclosed with double quotes
  - Backslash escaping sequences
  - Example: `"a \n b \" c \\ d"`
- Number
  - Decimals integers or floats
  - Example: `1, -0.5, 1.5e3`
- **Nested Object**
- **Nested Array**
- Boolean value: `true, false`
- Missing information: `null`

Keep in mind, JSON is NOT!

- Overly complex
- A "document" format
- A markup language
- A programming language

{ FRIENDS DON'T LET REAL FRIENDS USE RELATIONAL DATABASES }
mongoDB
14' x 48'
001636
CLEARCHANNEL
#1151277

MongoDB

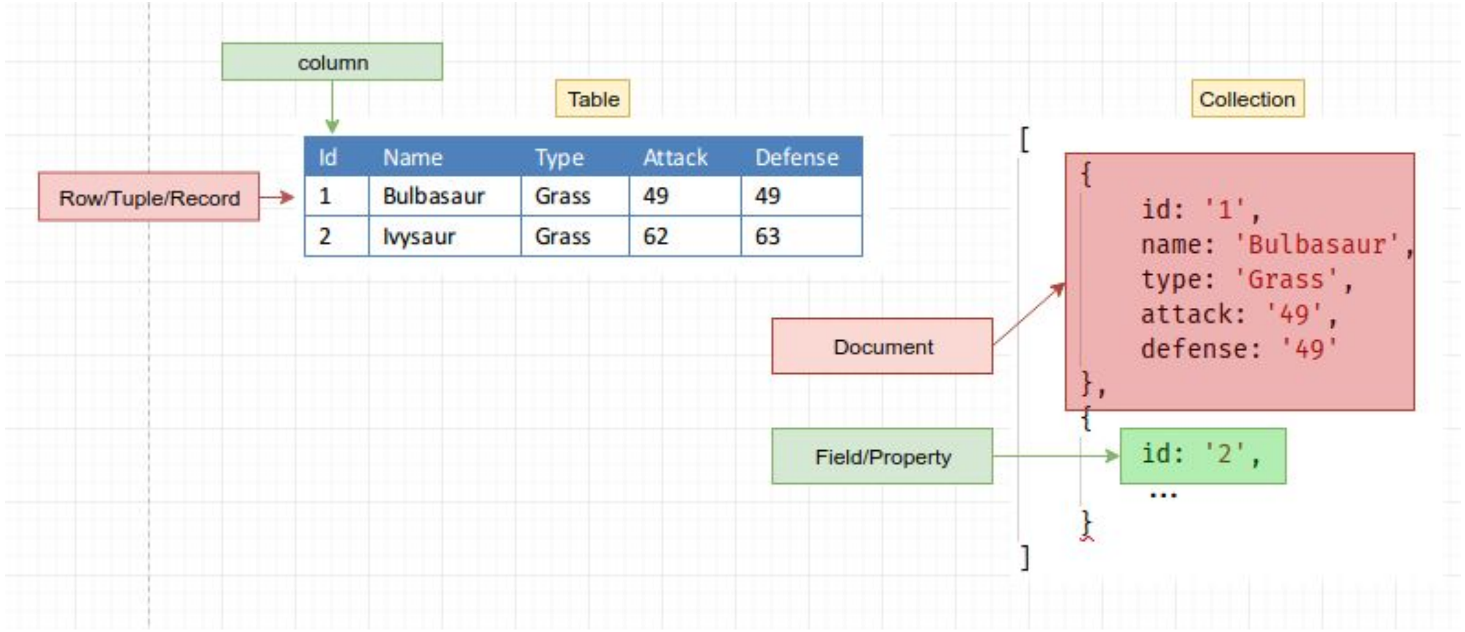
- **JSON document database**
- Features
  - Open source, high availability, eventual consistency, automatic sharding, master-slave replication, automatic failover, secondary indices
- Developed by MongoDB
- Implemented in C++, C, and JavaScript
- Operating systems: Windows, Linux, Mac OS X ...
- Initial release in 2009

MongoDB - data model

Database system structure: **Instance → databases → collections → documents**

- Database
- Collection
  - Collection of documents, usually of a similar structure
- Document
  - MongoDB document = one JSON object
  - Each document
    - belongs to exactly one collection
    - has a unique identifier **`_id`**

MongoDB - data model



Id	Name	Type	Attack	Defense
1	Bulbasaur	Grass	49	49
2	Ivysaur	Grass	62	63

```
[
  {
      id: '1',
      name: 'Bulbasaur',
      type: 'Grass',
      attack: '49',
      defense: '49'
  },
  {
      id: '2',
      ...
  }
]
```

SQL vs MongoDB concepts

SQL Terms/concepts	MongoDB terms
Database	Database
Table	Collection
Row	Document or BSON Document
Column	Field
Index	Index

SQL vs MongoDB concepts

SQL Terms/concepts	MongoDB terms
Table Joins	Embedded Documents and Linking
Primary Key	Primary Key
Transaction Begin, Commit/Rollback	NA
Schema	NA

MongoDB - data model

MongoDB document = one JSON object

- **Internally stored as BSON (Binary JSON)**
- Maximal allowed size: 16 MB (in BSON)
    - GridFS can be used to split large files into smaller chunks
- Restrictions on field names
    - _id (at the top level) is reserved for a primary key
    - **Field names cannot start with $**
        - Reserved for query operators
    - **Field names cannot contain .**
        - Used when accessing nested fields

MongoDB - data model

Primary keys

- Features of identifiers
  - **Unique** within a collection
  - **Immutable** (cannot be changed once assigned)
  - Can be of **any type** other than an array
- Design of identifiers
  - Natural identifier
  - Auto-incrementing number – not recommended
  - UUID (Universally unique identifier)
  - **ObjectId - special 12-byte BSON type** (default option)
    - Small, likely unique, fast to generate, ordered, based on a timestamp, machine id, process id, and a process-local counter

JSON-style Documents represented as BSON

```
{"hello":"world"}
```

```
\x16\x00\x00\x00\x02hello\x00\x06\x00\x00
\x00world\x00\x00
```

MongoDB - design questions

Flexible schema

- No document schema is provided, nor expected or enforced
    - **However, documents within a collection are similar in practice**
    - **MongoDB document = one JSON object**
        - I.e. even a complex JSON object with other recursively nested objects, arrays or values
- Design challenge
    - Balancing application requirements, performance aspects, and data retrieval patterns,
    - UUID (Universally unique identifier)
    - while considering structure of data and mutual relationships

Two main concepts: references vs. embedded documents

MongoDB - why use JSON?

JSON is open, human and machine-readable standard, and it's supports all basic data types: numbers, strings, boolean values, arrays and hashes.

More importantly,

- No need for a fixed schema. Just add what you want
- Format is programmer friendly. Unlike you know... XML!
- MongoDB uses BSON behind the scenes which is an extended format for JSON
- MongoDB reuses a Javascript engine for their queries. Makes complete sense in the world to re-use JSON for object representation

MongoDB - list databases

It’s strange to listen but true that MongoDB doesn’t provide any command to create databases. Then the question is how would we create database ?. The answer is – We don’t create database in MongoDB, we just need to use database with your preferred name and need to save a single record in database to create it. Let's check the current databases in our system.

```
> show dbs

admin     0.000GB
local     0.000GB
```

MongoDB - use new database

Now if we want to create database with name CSC424. The keyword **use** will instantiate database object. However, MongoDB doesn’t create any database yet, until you save something inside.

```
> use CSC424

switched to db CSC424
```

MongoDB - create a collection

MongoDB stores the data in the form of JSON documents. A group of such documentation is collectively known as a collection in MongoDB. Thus, a collection is analogous to a table in a relational database while a document is analogous to a record. To store documents, we first need to create a collection. The exciting thing about a NoSQL database is that unlike SQL database, you need not specify the column names or data types in it.

```
> db.createCollection("students")

{ "ok" : 1 }
```

MongoDB - show collections

If you fancy to see the collections you have it in the current db instance, use the following command.

```
> show collections

students
```

MongoDB - drop database

MongoDB provides `dropDatabase()` command to drop currently used database with their associated data files. Before deleting make sure what database are you selected using db command.

```
> db

students

> db.dropDatabase()

{ "dropped" : "CSC424", "ok" : 1 }
```

QA