NoSQL designing & MongoDB

CS415 – Relational and Document-Based Databases

Computer Science Department

Maharishi International University

Maharishi International University - Fairfield, Iowa

Content

- What is NoSQL?
- NoSQL databases benefits and types
- How does NoSQL work?
- Partition (Primary) key
- JSON, XML, and BSON
- When to use NoSQL or SQL?
- NoSQL DB Designing
- Create
- Find, FindOne

What are NoSQL databases?

NoSQL database is a mechanism for storage and retrieval of data that is modeled **other than the tabular relations** used in relational databases.

NoSQL databases are widely recognized for

- their ease of development
- performance at scale.

NoSQL databases break the traditional mindset of storing data at a single location. Instead, NoSQL distributes and stores data over a set of **multiple servers**.

Why should you use a NoSQL database?

Scalability: NoSQL databases are generally designed to scale out by using distributed clusters of hardware instead of scaling up by adding expensive and robust servers.

High-performance: NoSQL databases provide 2-digit millisecond performance.

Flexibility: NoSQL databases generally provide flexible schemas that enable faster and more iterative development. The flexible data model makes NoSQL databases ideal for semi-structured and unstructured data.

Other advantages of NoSQL database

- **More reliable** by employing many servers
- **Significantly cheaper** to scale with commodity hardware
- **Less Management** than RDBMS
- **Unlimited Space** with cloud technologies

More performant

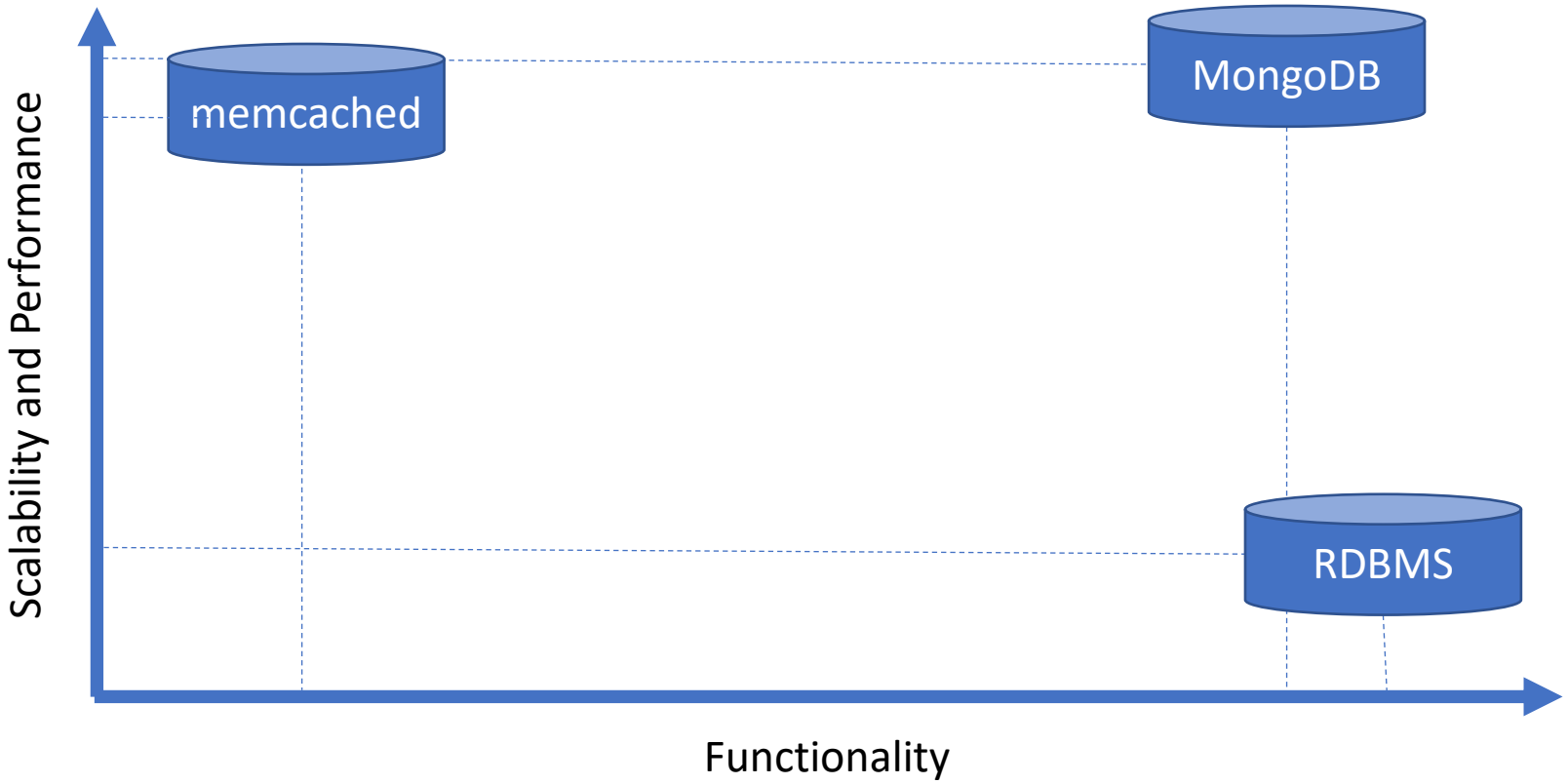
Looks like under heavy load, MongoDB is about 100 times faster.

```
MongoDB Client
Warming up ...
Building insert data...
Waiting on mutex
Running!
Finished with 10000 MongoDB inserts in 2.032 sec.
Done
```

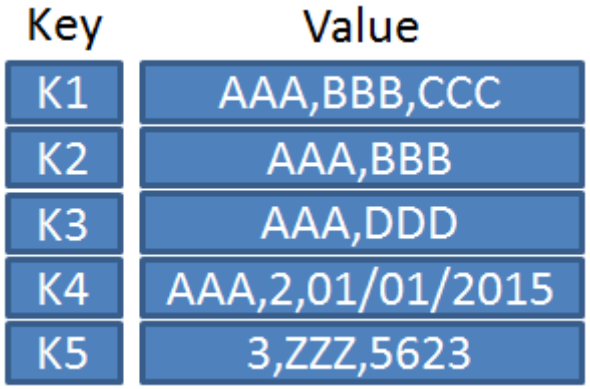
```
SQL Server Client
Warming up ...
Building insert data...
Waiting on mutex
Running!
Finished with 10000 SQL inserts in 204.215 sec.
Done
```

That’s right. It’s 2 seconds verses 3.5 minutes!

Databases comparison

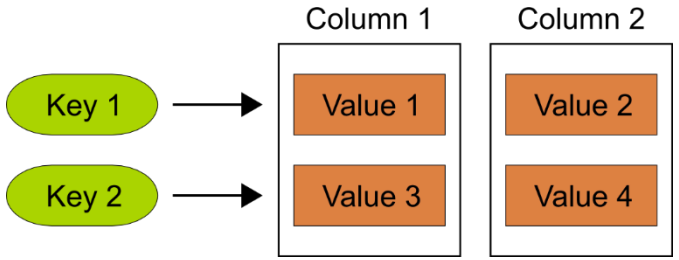


NOSQL Database Types



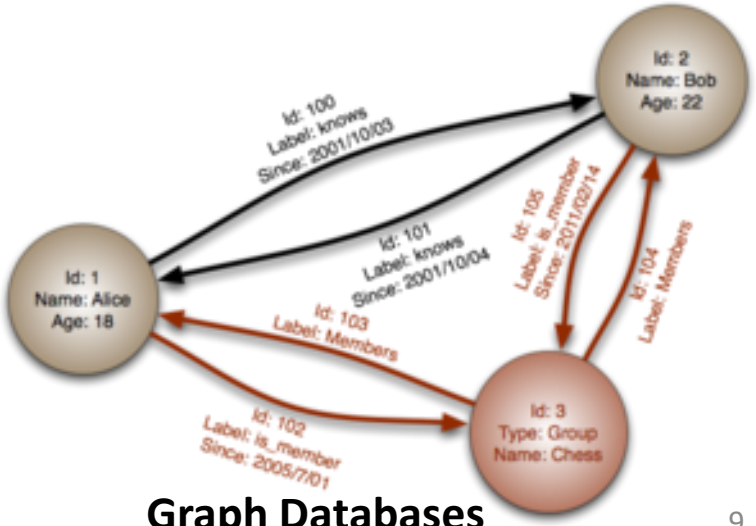
Key-Value Stores

Document Databases



Column Family Stores

Key-Value pairs in hash table, always unique key. Logical group of keys are called: buckets.
Document Databases uses Key-Value pairs in a document (JSON, BSON).
Column Stores data is stored in cells that are grouped in columns rather than rows (unlimited columns).
Graph Databases, uses flexible graphical representation (edges and nodes). Very fast for finding deep connections.



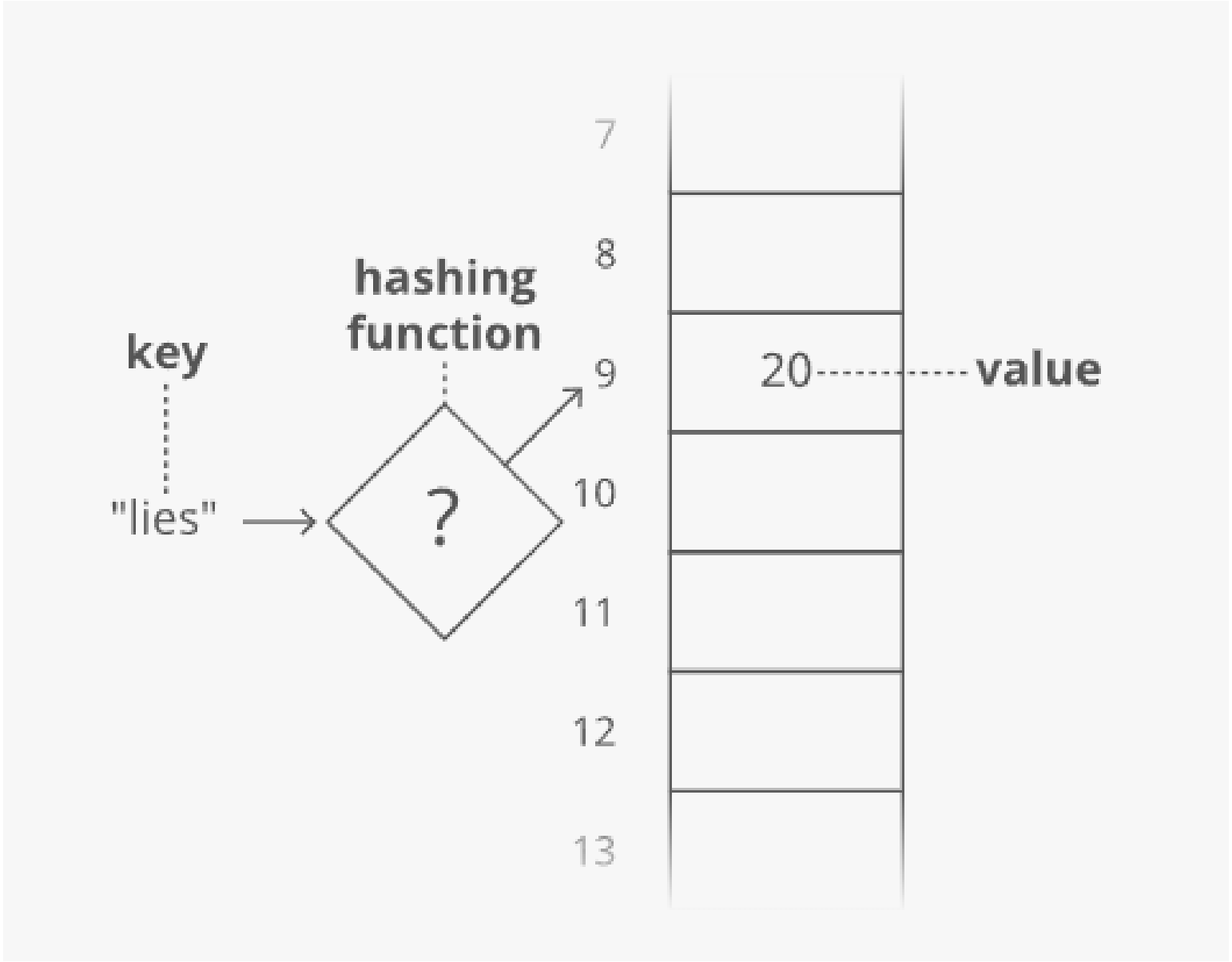
Graph Databases

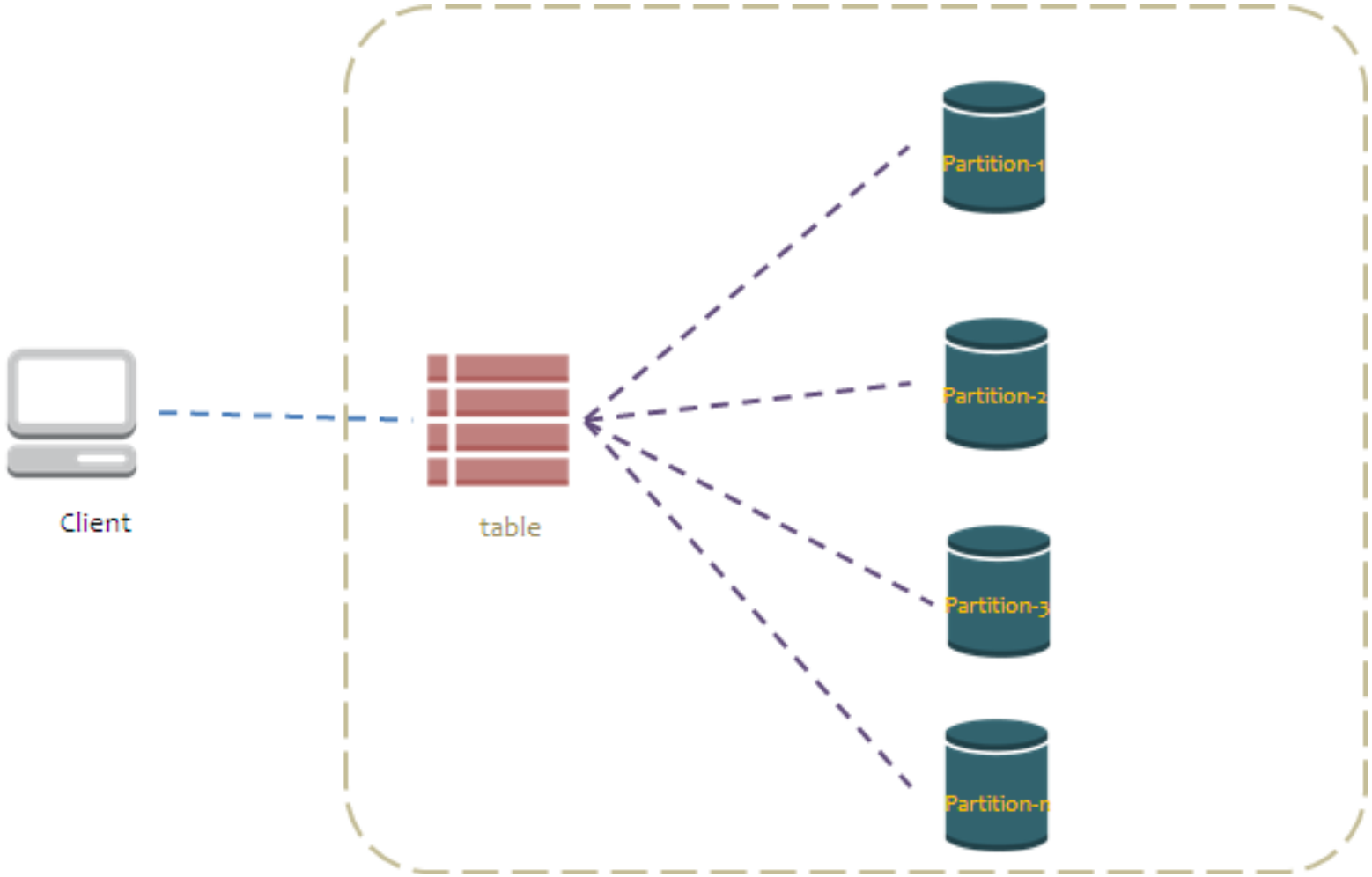
How does NoSQL database work?

Similar to the HashTable data structure. A hash table is a type of data structure that stores key-value pairs. The key is sent to a hash function that generates the hash that determines where data is stored.

Performance:

- Insert O(1)
- Delete O(1)
- Update O(1)
- Lookup by key O(1)
- Lookup by index O(log N)
- Lookup by other attributes O(n)

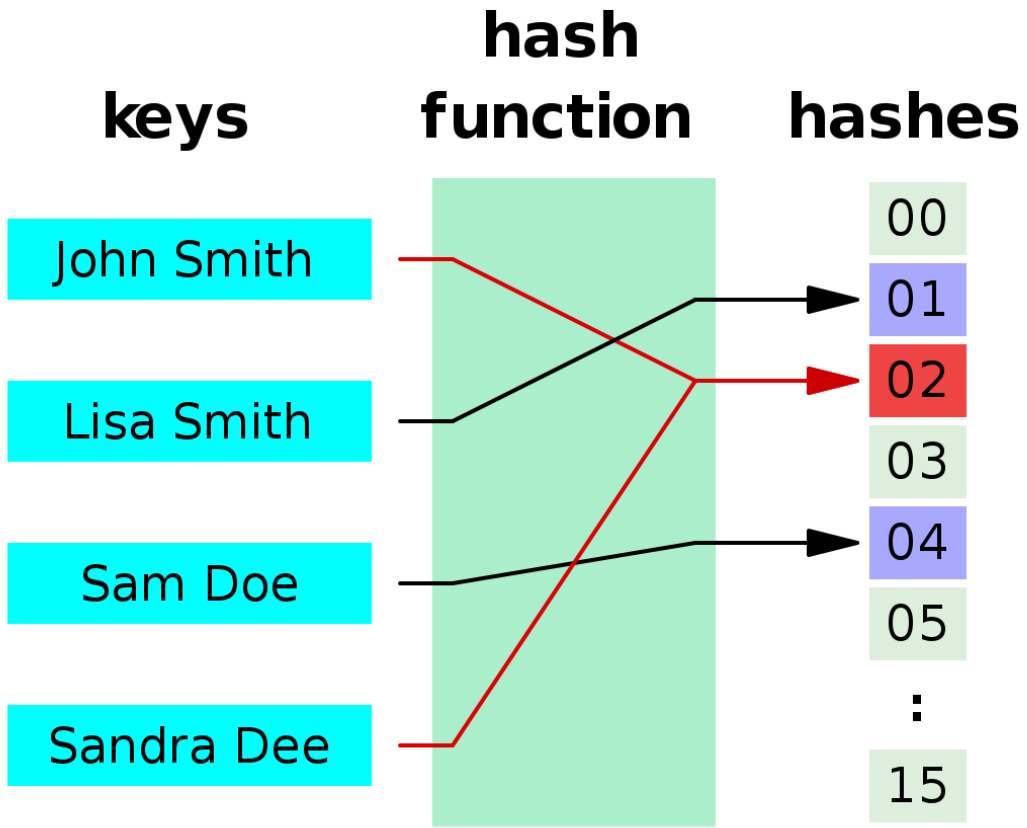

key
"lies"
hashing
function
?
7
8
9
10
11
12
13
20
value

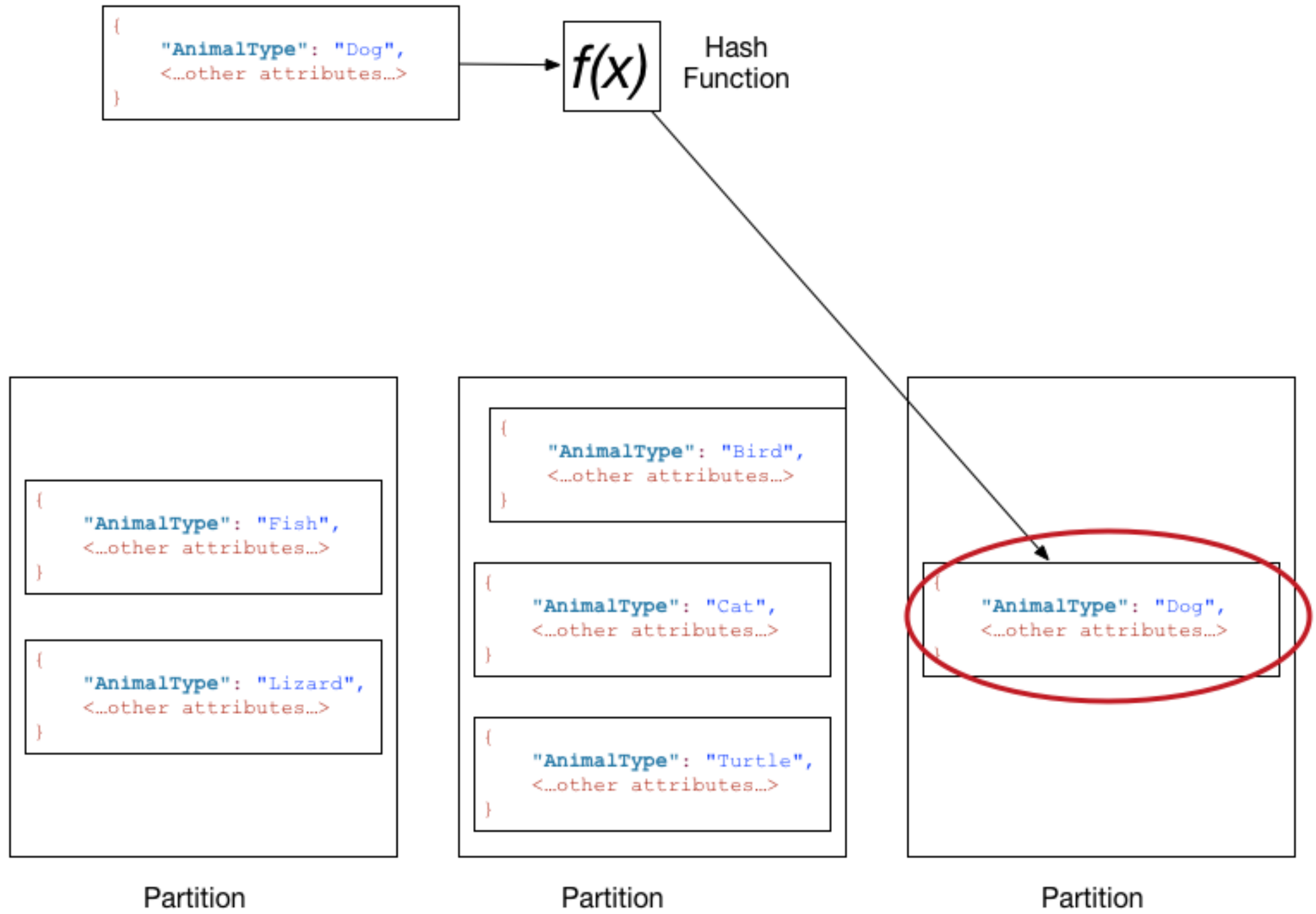

Client
table
Partition-1
Partition-2
Partition-3
Partition-n

Partition (Primary) key

It is a simple primary key.

NoSQL databases use the partition key's value as input to an internal **hash function**. The output from the hash function determines the partition in which the item will be stored.




{
"AnimalType": "Dog",
<…other attributes…>
}
f(x)
Hash Function
{
"AnimalType": "Fish",
<…other attributes…>
}
{
"AnimalType": "Lizard",
<…other attributes…>
}
{
"AnimalType": "Bird",
<…other attributes…>
}
{
"AnimalType": "Cat",
<…other attributes…>
}
{
"AnimalType": "Turtle",
<…other attributes…>
}
{
"AnimalType": "Dog",
<…other attributes…>
}
Partition
Partition
Partition

MongoDB is a JSON-like document DB categorized in NoSQL. Stores data in JSON.

- **Documents:** The records in the DB.
- **Collections:** Grouping documents, like a table in SQL.
- **Replica Sets:** Ensuring High Availability by copying data onto another server.
- **Sharding:** Horizontal scaling to Handle Massive Data Growth
- **Indexes:** Improving Query Speed

Learn more: MongoDB basics

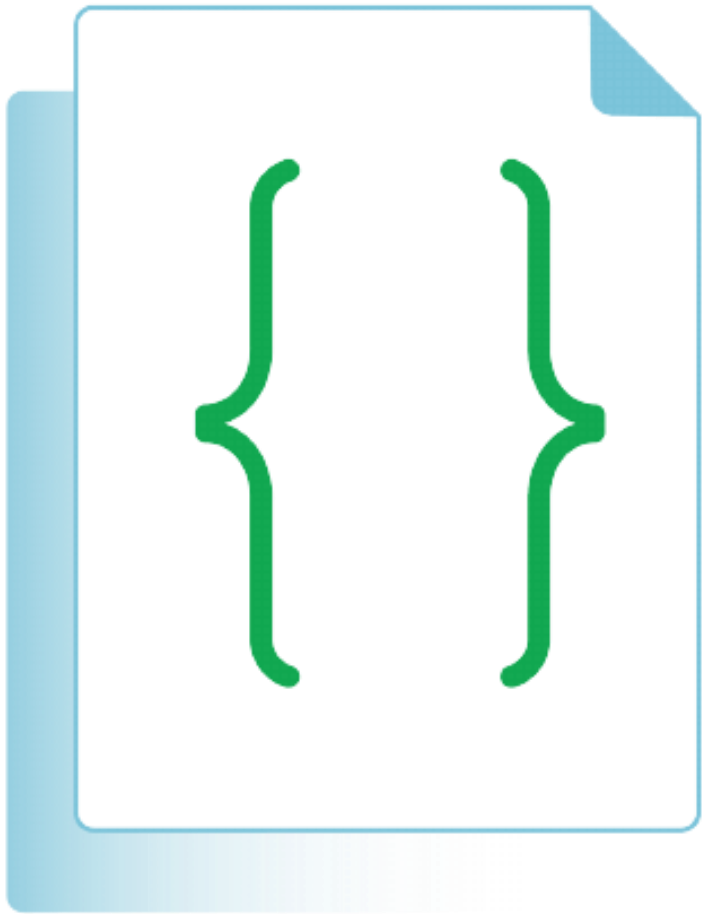
What is JSON?

JSON stands for JavaScript Object Notation. It is a JavaScript object contains data wherein a string key is mapped to a value.

Value can be:

- Number
- String
- Function
- Another object (Array)

JSON is now used everywhere and quickly overtook XML.



```
{
    "model": "Volvo C70",
    "year": 2007,
    "bodyStyle": [
        "coupe",
        "convertible"
    ],
    "engine": {
        "model": "D5",
        "power": "178hp"
    }
}
```

What is XML?

Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human and machine readable like HTML.

XML is more difficult than JSON for a human to read, significantly **more verbose**, and less ideally suited to representing object structures used in modern programming languages.

But there are still many legacy systems out there that uses XML so it is worth knowing what that is.

XML Vs JSON

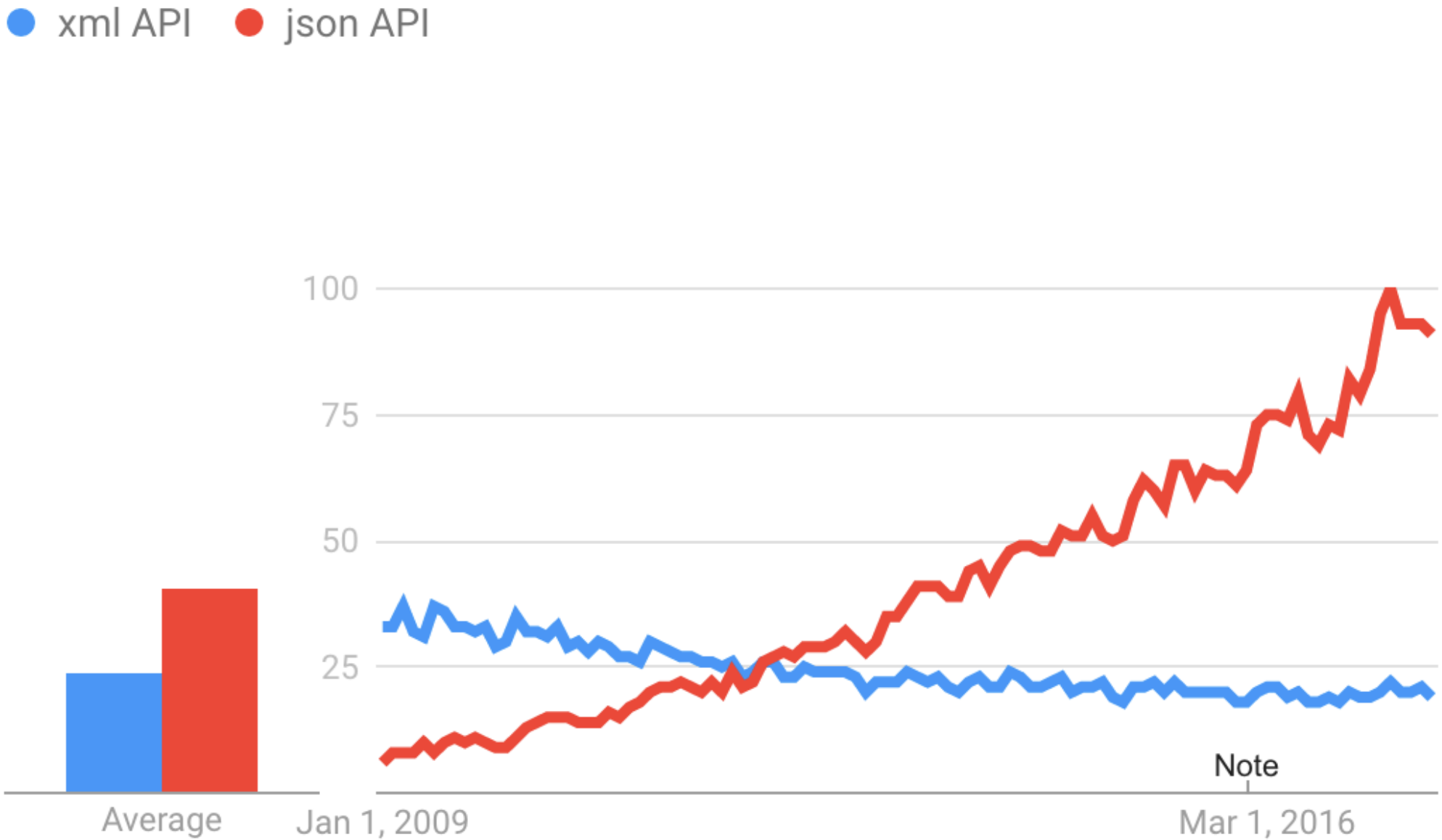
```xml
<users>
    <user>
        <name>John</name>
        <age>22</age>
    </user>
    <user>
        <name>Mary</name>
        <age>21</age>
    </user>
</users>
```

```json
var users=[
        {
            name:"John",
            age:22
        },
        {
            name:"Mary",
            age:21
        }
    ];
```

JSON	XML
Text based format (Not a Language)	Markup Language
Free to define anything	Has some rules
Smaller Size	Big in Size due to markups
JSON is similar to Java script Objects literals. Browser read faster.	Browser need parsers to handle XML. Slow processing.
No support on namespaces and comments	Both are supported.

Interest over time

Google Trends



BSON

BSON is binary JSON. There are 3 advantages of using BSON over JSON in NoSQL databases.

1. JSON is slower
2. JSON takes more space in the storage
3. JSON supports a limited number of data types

Extra datatypes that BSON supports are

- Number (Integer, Float, Long, Decimal128...)
- Raw binary
- Date (Developer like to use Unix timestamp)

Read more: JSON and BSON

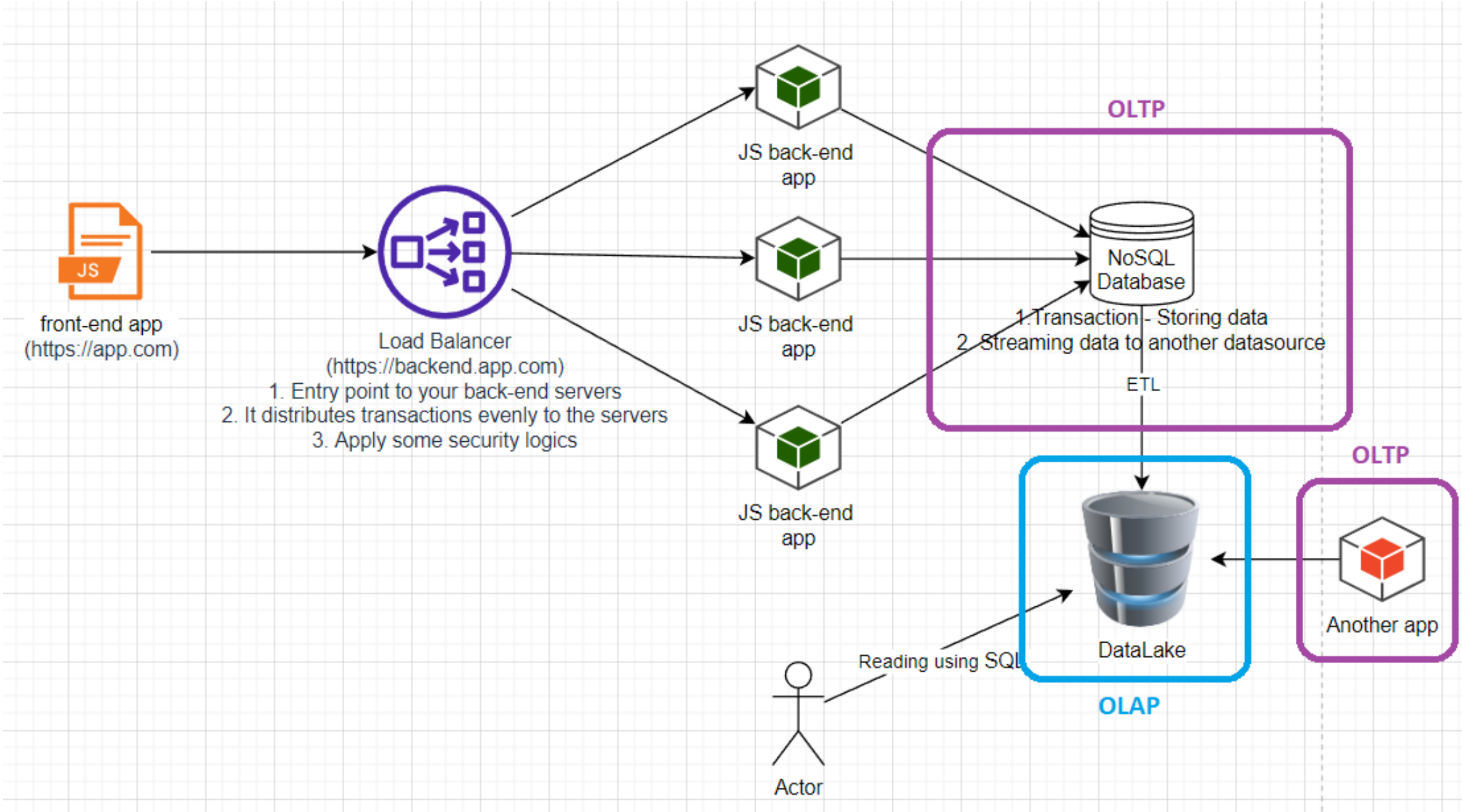
What is Unix timestamp?

The Unix timestamp is a way to track time as a running total of seconds. This count starts on Jan 1st, 1970 and ends on Jan 19th, 2038 due to a 32-bith overflow.

You can easily take the timestamp in JS using Math.floor(Date.now() / 1000) in seconds.

OLTP vs OLAP

- **OLTP** (Online transaction processing) – The app software developers write. Used for day-to-day operations.
- **OLAP** (Online analytical processing) – applies complex queries to large amounts of historical data, aggregated from OLTP databases and other sources, for data mining, analytics, and business intelligence projects.
- The data from one or more OLTP databases is ingested into OLAP systems through a process called extract, transform, load (**ETL**).
- For more: OLTP and OLAP: a practical comparison


JS
front-end app
(https://app.com)
Load Balancer
(https://backend.app.com)
1. Entry point to your back-end servers
2. It distributes transactions evenly to the servers
3. Apply some security logics
JS back-end app
JS back-end app
JS back-end app
OLTP
NoSQL Database
1.Transaction - Storing data
2. Streaming data to another datasource
ETL
OLTP
Another app
DataLake
OLAP
Reading using SQL
Actor

SQL vs NoSQL

When to use NoSQL?

- Transactional applications; **OLTP** (Online transaction processing)
- You need hyper scale & low latency
- Flexible data models
- Very large amounts of data and very large numbers of users

When to use SQL?

- Ad hoc queries (complex queries); data warehousing; **OLAP** (online analytical processing)
- Your data is predictable and highly structured
- Your workload volume is consistent
- You need full control on your database and managing high availability, scalability yourself

SQL vs NoSQL

SQL	NoSQL
Data uses Schemas	Schema-less
Relations	No (or very few) Relations
Data is distributed across multiple tables	Data is typically merged /nested in a few collections
Horizontal scaling is difficult / impossible; Vertical scaling is possible	Both horizontal and vertical scaling is possible
Limitations for lots of (thousands) read & write queries per second	Great performance for mass read & write requests

Document Data Model

- A record in MongoDB is a **Document**
- Structure of key/value pairs
- Values may contain other documents (embedded documents), arrays and arrays of documents (rich document).

```
{
   _id: 1,
   name: "Asaad",
   email: "asaad@mum.edu",
   courses: ["CS472", "CS572"]
}
```

Document Implications

- Embedded documents & arrays reduce the need for expensive joins.
- Atomic transaction on the document level. Atomic transaction is either all occur or nothing occur.

Data Types

The value of a field can be any of the **BSON data types**, including other documents, arrays, and arrays of documents.

```
var mydoc = {
        _id: ObjectId("5099803df3f4948bd2f98391"),
        name: { first: "Asaad", last: "Saad" },
        birth: new Date('Oct 31, 1979'),
        courses: [ "CS472", "CS572" ],
        students : NumberLong(1250000)
}
```

Schema and Agile Structure

The documents in a single collection do not need to have the same set of fields and the data type for a field can differ across documents within a collection.

To add more information to a table in relational DBMS, we need to add more columns to the table (which might leave many null values) or add new table.

MongoDB is agile, there will be no need to have same structure in documents. Every document can have its own structure.

(Starting of MongoDB 3.2, you can enforce document validation rules for a collection during update and insert operations)

Collections

MongoDB stores documents in **collections**. *(Collections are similar to tables in relational databases. When designing the collection, think it as a view in RDBMS.)*

If a database/collection does not exist, MongoDB creates the db/collection when you first store data for that collection

```
use myNewDB

db.myNewCollection.insert( { x: 1 } )
```

The insert() operation creates both the database myNewDB and the collection myNewCollection if they do not already exist.

Exploring the shell - Demo

```
show dbs
use testDB // switch or create
show collections
db.testCol.insert({"name": "Saad"})// db var refers to the current database
db.testCol.find() // notice _id
// passing a parameter to find a document that has a property "name" and value
"Asaad"
db.testCol.find({"name":"Asaad"})
// save() = upsert if _id provided
db.testCol.save({"name":"Mike"})
// insert 10 documents – Shell is C++ app that uses V8
for (var i=0; i<10; i++){ db.testCol.insert({"x": i}) }
```

Exploring the shell - Demo

```
db.testCol.save({a:1, b:2})
db.testCol.save({a:3, b:4, fruit: ["apple", "orange"] })
db.testCol.save({name: "Asaad", address: {city: "Fairfield",
                                          zip: 52557,
                                          street: "1000 N 4th street"} })
// show documents in a nice way, it will only work when you have nested or larger
documents:
db.testCol.find().pretty()
```

General Rules

- Field names are strings. You can ignore quotes.
- The field name `_id` is reserved for use as a primary key. It is immutable and always the first field in the document. It may contain values of any BSON data type.
- The field names cannot start with the dollar sign ($) character and cannot contain the dot (.) character or `null`. Field names cannot be duplicated.
- The maximum BSON document size is 16 megabytes. *(To store documents larger than the maximum size, MongoDB provides the GridFS API)*

Modeling Introduction

```
// posts collection
{ title: '',
  body: '',
  user: '', // no need for ID
  date: '',
  comments: [ {user: '', email: '', comment: ''},
              {user: '', email: '', comment: ''}]
  tags: ['', '', ''] }

// authors collection
{ user:'',
  password:''}
```

Why did we embed *tags* or *comments*? Rather than have them in separate collection? Because they need to be accessed at the same time we access the *post*. We don't need to access *comments* or *tags* independently without accessing the *post*.

MongoDB Schema Design

In MongoDB we use **Application-Driven Schema**, which means we design our schema based on **how we access the data**.

Note: The only scenario we cannot embed is when data exceeds 16 MB and we need to put it in separate collection or another cloud object storage service.

Design Considerations

Posts

```
{ _id,
user_id
title,
body,
shares_no
date }
```

users

```
{ _id,
name,
email,
password }
```

comments

```
{ _id,
user_id,
post_id
comment_text
order }
```

post_likes

```
{ post_id,
user_id }
```

Remember that we don't have constrains, so this design will not work as we need to perform too much work (4 joins) in the code to retrieve our data

Design Considerations

```
{
  _id: objectId(),
  user: 'user1', // use it as ID
  title: '',
  body: '',
  shares_no: 0,
  date: ,
  comments: [
     {user:'user2', comment_text:''},
     {user:'user3', comment_text:''}]
  likes: [ 'user1', 'user2']
}
```

This design is optimized for data access pattern so we can access the information much faster with 1 query. Especially that there is no need for data to be updated later.

Transactions vs Atomic Operations

- In the world of **relational DB**, our data is usually located in many tables, so in order to update something, we will need to access all these tables and perform the update on all of them in one **Transaction**.
- In **MongoDB**, our data is usually located in one document, and because most operations are **Atomic**, we accomplish the same thing without the need for transactions.
- Try to restructure your design to use **less documents as possible** (with one document we guarantee Atomic operations)

To embed or not to embed

One-to-One relation

- Linking is fine
- Embed in either sides is fine
- Embed in two side is fine

employee: resume
building: floor plan
patient: medical history

One to Many relation

- Use linking when large data and have separate collections
- Use Embed when **One-to-Few**

city: people
post: comments

Many to Many relation

- Embed is better with **Few-to-Few**
- Linking is better for performance in large data

books: authors
students: teachers

The only scenario we cannot embed is when data exceeds 16 MB and we need to put it in separate collection.

MongoDB in the Cloud

Atlas is a fully managed cloud database service featuring automated provisioning and scaling of MongoDB databases.

Atlas's Database-as-a-Service platform powers databases across AWS, Azure, and Google Cloud Platform.

Other options:

- Use Mongo Docker container

CRUD in MongoDB

There is no special SQL language to perform CRUD in MongoDB.

Many CRUD operations exist as methods/functions on objects in programming language API, NOT as separate language.

CRUD	MongoDB	SQL
Read	`find()`	`select`
Create	`insert()`	`insert`
Update	`update()`	`update`
Delete	`remove()`	`delete`

Searching an Object

```
{_id: 1, email: { work: "work@mum.edu", personal: "personal@gmail.com"} }

// nothing will be returned
db.col.find({ email: { work: "work@mum.edu" } })
db.col.find({ email: { personal: "personal@gmail.com" } })
db.col.find({ email: { personal: "personal@gmail.com", work: "work@mum.edu"} })

// will work
db.col.find({ email: { work: "work@mum.edu", personal: "personal@gmail.com"} })

// how to search for one key only?
db.col.find({ "email.work": "work@mum.edu" })
db.col.find({ "email.personal": "personal@gmail.com" })
```

`db.collection.findOne({query}, {projection: {} })`

Returns **one document** that satisfies the specified **query** criteria. If multiple documents satisfy the query, this method returns the first document according to the **natural order** which reflects the order of documents on the disk. If no document satisfies the query, the method returns null.

- The **`query`** is equivalent to **`where`** in SQL, it takes the form of JSON object.
- The **`project`** method accepts JSON of the following form:

```
{ field1: <boolean>, field2: <boolean> ... }
```

Examples - findOne()

```
// return one document with all fields
db.collection.findOne({})

// return one document with two fields "_id" and "name"
db.collection.findOne({}, { projection: {name: 1} )

// return one document that has "name" property with value "Asaad",
   this document will have all fields but "_id" and "birth"
db.collection.findOne({name: 'Asaad'},{ projection: { _id: 0, birth: 0 } })
```