Lecture 19-NoSQL Data Model & Data Formats

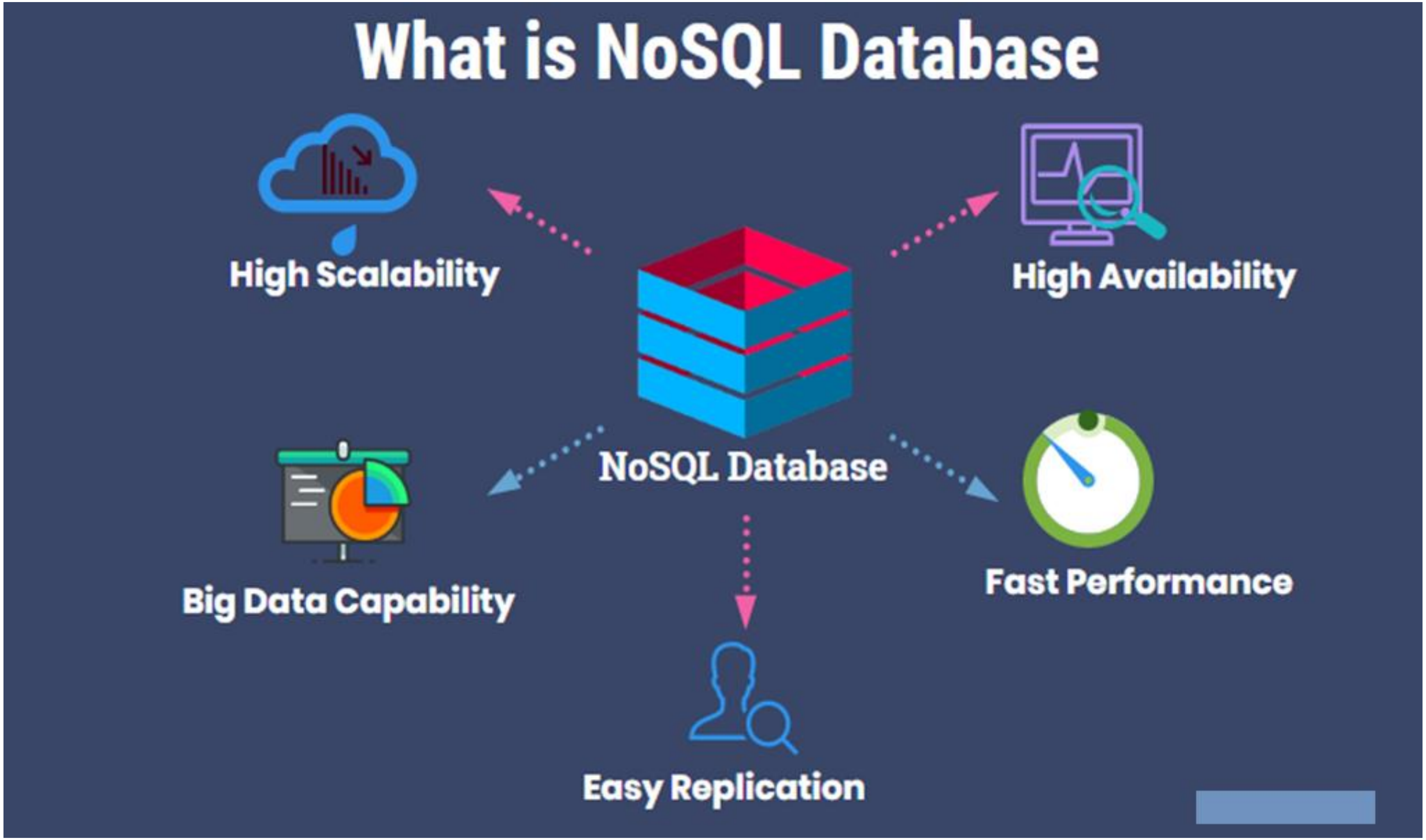
CS211 - Introduction to Database

NoSQL Data Model

What is it? NoSQL

Not only SQL or actually **Not Relational** database.

- It does not require a **fixed schema**.
- It **avoids joins**, and hence, is easy to scale.
- Its major purpose is for **distributed data stores** with humongous data storage needs.
- NoSQL is used for **Big data** and **real-time web apps**, e.g. Twitter, Facebook and Google.


What is NoSQL Database
High Scalability
High Availability
NoSQL Database
Big Data Capability
Fast Performance
Easy Replication

Data stored: RDBMS vs. NoSQL

- Traditional RDBMS requires **fixed schemas**. The stored data is **structured**.

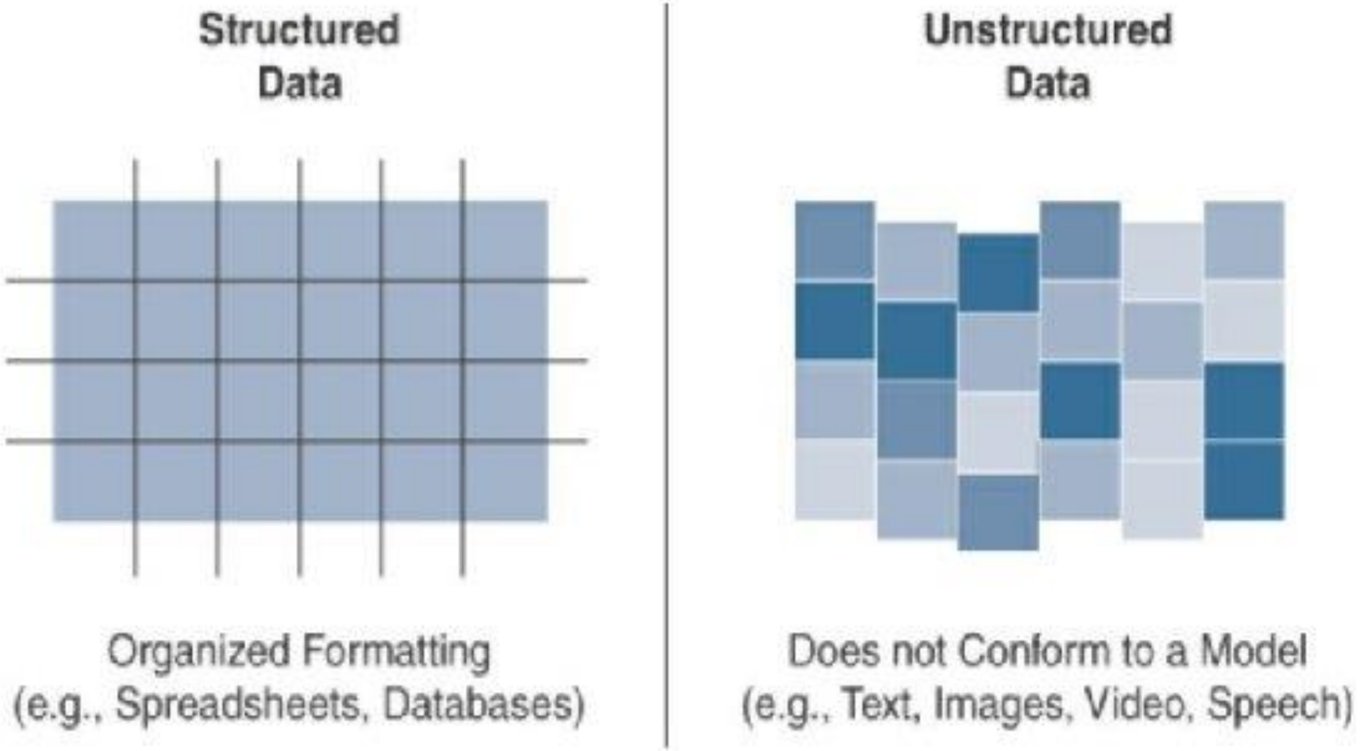
- NoSQL database system can store different types of data:

1. **Structured data**
   - ❑ Comprised of **clearly defined data types with patterns** that make them easily searchable.
   - ❑ Example- relational database, spreadsheet etc.

2. **Unstructured data**
   - ❑ **Does not have a predefined data model**. It comprises of data that is usually not as easily searchable.
   - ❑ Example- audio, video, and social media postings etc.

Structured Versus Unstructured Data

- Require different toolsets to analytics Structured data and unstructured data.



Data stored: RDBMS vs. NoSQL

3. Semi-structured data

- It does not have a predefined data model and is more complex than structured data, yet easier to store than unstructured data.
- It is normally considered unstructured data, but that also has **metadata** that identifies certain characteristics.
- Metadata is **'data about data'**. It's a small portion of a file that contains data about the contents of the file.
- Example- JSON, CSV, XML, TCP/IP packets

Data stored: RDBMS vs. NoSQL

4. Polymorphic data

- In the case of database, polymorphism is the ability of a **single column or field to contain data of different types**.
- Polymorphic data means that in **one collection** you have **many versions of document schema** (e.g. different field type, fields that occur in some documents etc.).

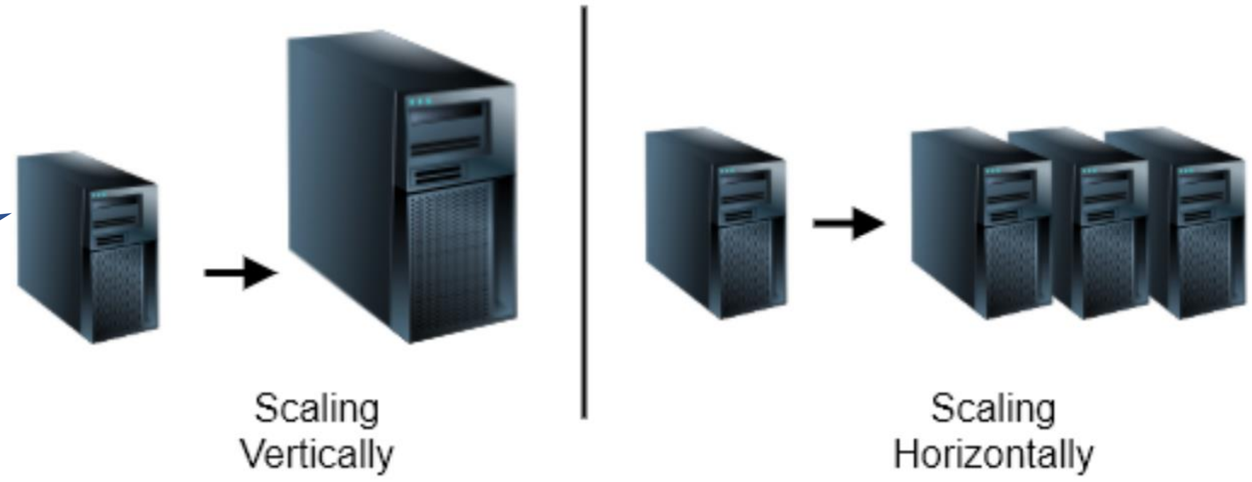
```
{
        "user" : "James",
        "email" : "james@gmail.com"
        "age" : 39
},

{
        "user" : "Anna",
        "email" :  [
                        "anna@gmail.com"        ,
                        "anna@Hotmail.com"
                   ]
}
```

Why NoSQL?

- The concept of NoSQL databases became popular with Internet giants like Google, Facebook, Amazon, etc. who deal with **huge volumes of data**.
- The **system response time** becomes slow when you use RDBMS for massive volumes of data.
- To resolve this problem, we could **"scale up"** our systems by upgrading our existing hardware. This process is expensive.

More CPU, RAM , HDD…

Process is expensive



More servers …

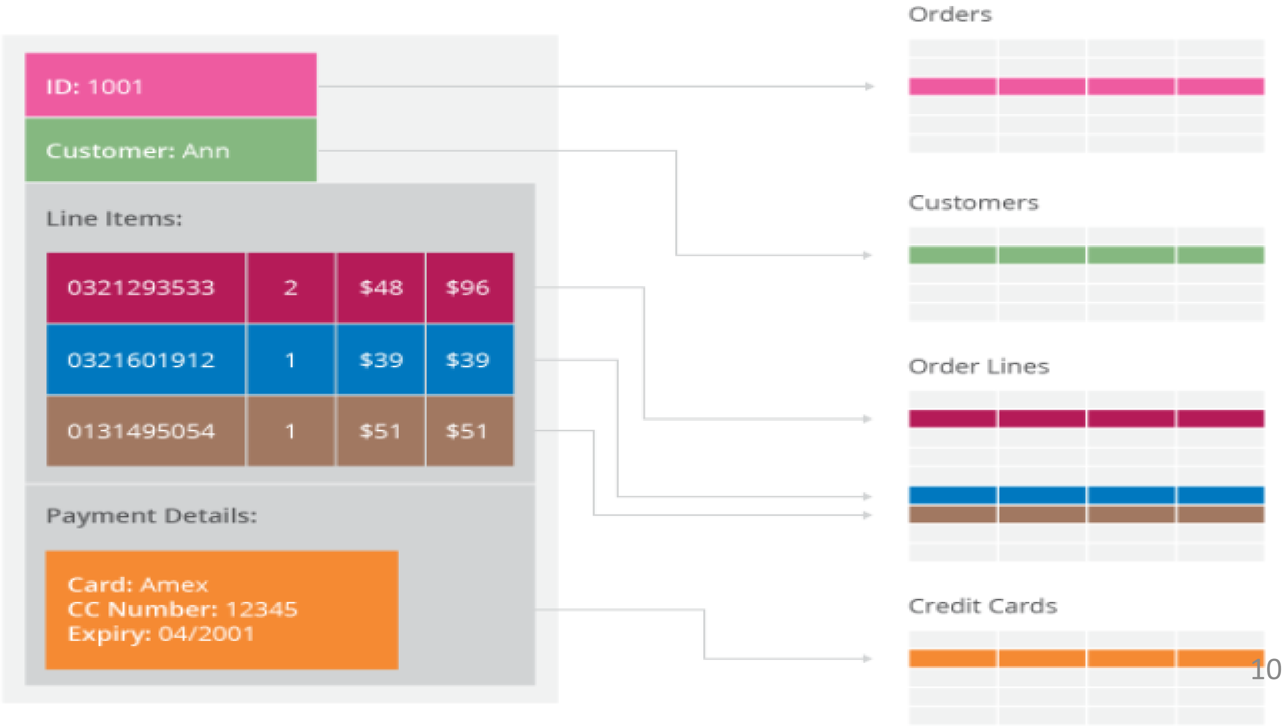
Process is flexible

NoSQL designed for Scaling Horizontally

Features of NoSQL database

❖ **Non-relational**

- **Never follow** the relational model.
- **Never** provide tables with **flat fixed-column records**.
- Work with **self-contained aggregates** (collections, documents, etc.).
- Doesn’t require **object-relational mapping** and **data normalization.**
- No complex features like query languages, query planners, referential integrity joins, ACID properties etc.



Features of NoSQL database

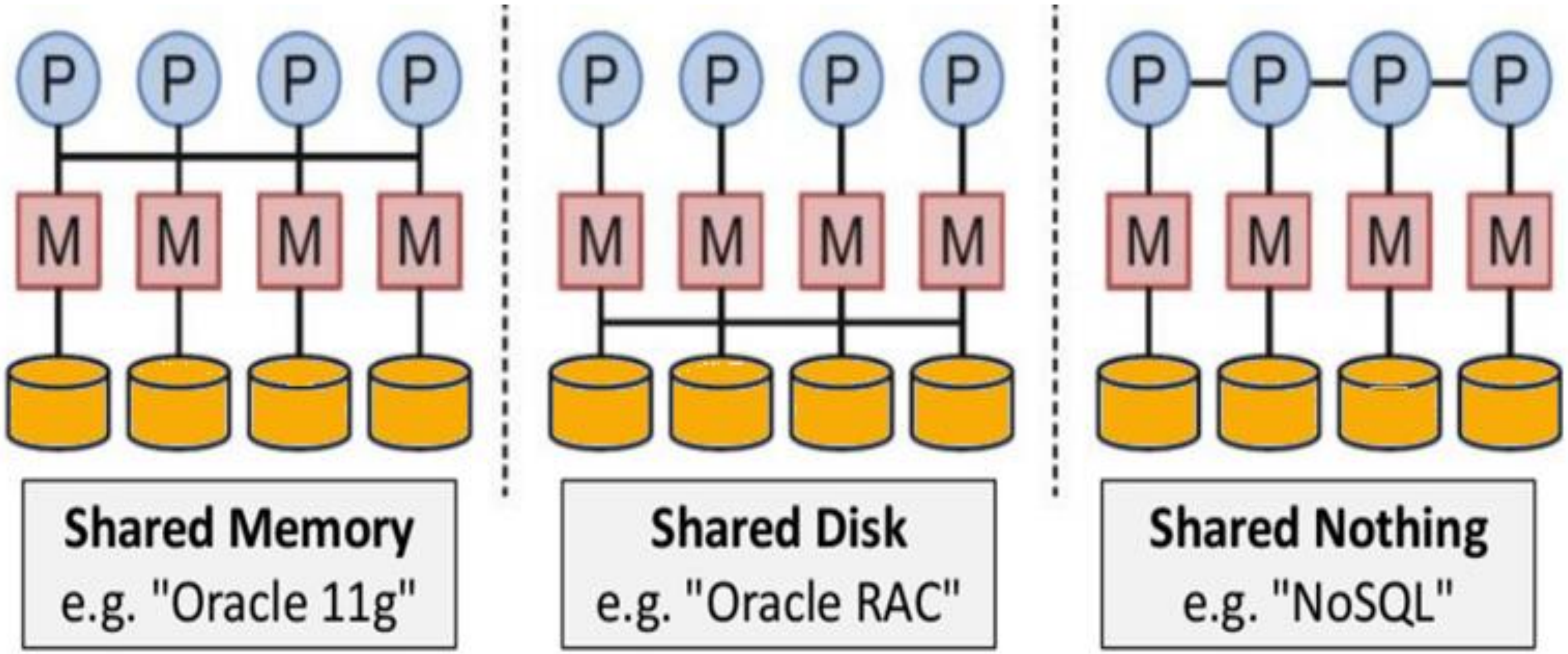
❖ **Schema-free**

- NoSQL databases are either **schema-free** or have **relaxed schemas.**
- Do not require any sort of **definition of the schema** of the data.
- Offers **heterogeneous** structures of data in the same domain.

❖ **Distributed**

- Multiple NoSQL databases can be executed in a distributed fashion.
- Offers **auto-scaling** and **fail-over** capabilities.
- Often ACID concept can be sacrificed for scalability and throughput.
- Provides **eventual consistency:** Eventual Consistency is a guarantee that when an update is made in a distributed database, that update will eventually be reflected in all nodes that store the data, resulting in the same response every time the data is queried.
- **Shared Nothing Architecture**. This enables less coordination and higher distribution.

NoSQL follows shared nothing architecture



Pros of SQL

- **Reduced data storage:** Normalization removed data redundancy and duplication
- **ACID-compliant:** Strong data integrity, security and consistency
- **Normalization:** Database engines are better at optimizing queries to fit on-disk representations
- **Great for complex queries:** SQL is efficient at processing queries and joining data across tables
- **Standardized language:** Across different RDBMS

Cons of SQL

- **Vertical scaling:** Relatively more expensive than horizontal scaling
- **Rigid data model:** Require up-front design, and harder to make changes (require data migrations and possibly downtime)
- **Single point of failure:** Mitigated by replication and failover techniques
- **Big data:** Fails to handle big data efficiently

Pros of NoSQL

- Handles big data which manages data velocity, variety, volume, and complexity
- No Single Point of Failure
- Easy Replication
- No Need for Separate Caching Layer
- It provides fast performance and horizontal scalability
- Can handle structured, semi-structured, and unstructured data with equal effect
- For Object-oriented programming languages, it is easy to use
- NoSQL databases don’t need a dedicated high-performance server
- Support Key Developer Languages and Platforms
- Simple to implement than using RDBMS
- It can serve as the primary data source for online applications
- Excels at distributed database and multi-data center operations
- Offers a flexible schema design which can easily be altered without downtime or service disruption

Cons of NoSQL

- No standardization rules
- Limited query capabilities
- NoSQL tools are not comparatively mature
- It provides vary limited traditional database capabilities, like consistency when multiple transactions are performed simultaneously
- When the volume of data increases it is difficult to maintain unique values as keys Doesn’t work as well with relational data
- The learning curve is stiff for new developers
- It is open source, so not so popular for enterprises due to lack of professional support & maintenance
- Less secure - Difficult to verify data integrity and consistency (only eventual consistency achieved)
- No normalization - Database engines are not as good at optimizing queries

When should we use NoSQL?

1. **Fast-paced Agile development**
    - The pace of development with NoSQL databases can be much faster than with a SQL database.
2. **Storage of structured and semi-structured data**
    - The structure of many different forms of data is more easily handled and evolved with a NoSQL database.
3. **Huge volumes of data**
    - The amount of data in many applications cannot be served affordably by a SQL database.
4. **Requirements for scale-out architecture**
    - The scale of traffic and need for zero downtime cannot be handled by SQL.
5. **Modern application paradigms like microservices and real-time streaming**
    - Cloud deployment of NoSQL supports micro-services and real-time streaming technologies.

RDBMS vs NoSQL: Data Modeling Example

An example of storing information about a user and their hobbies

Users

ID	first_name	last_name	cell	city
1	Leslie	Yepp	8125552344	Pawnee

Hobbies

ID	user_id	hobby
10	1	scrapbooking
11	1	eating waffles
12	1	working

RDBMS

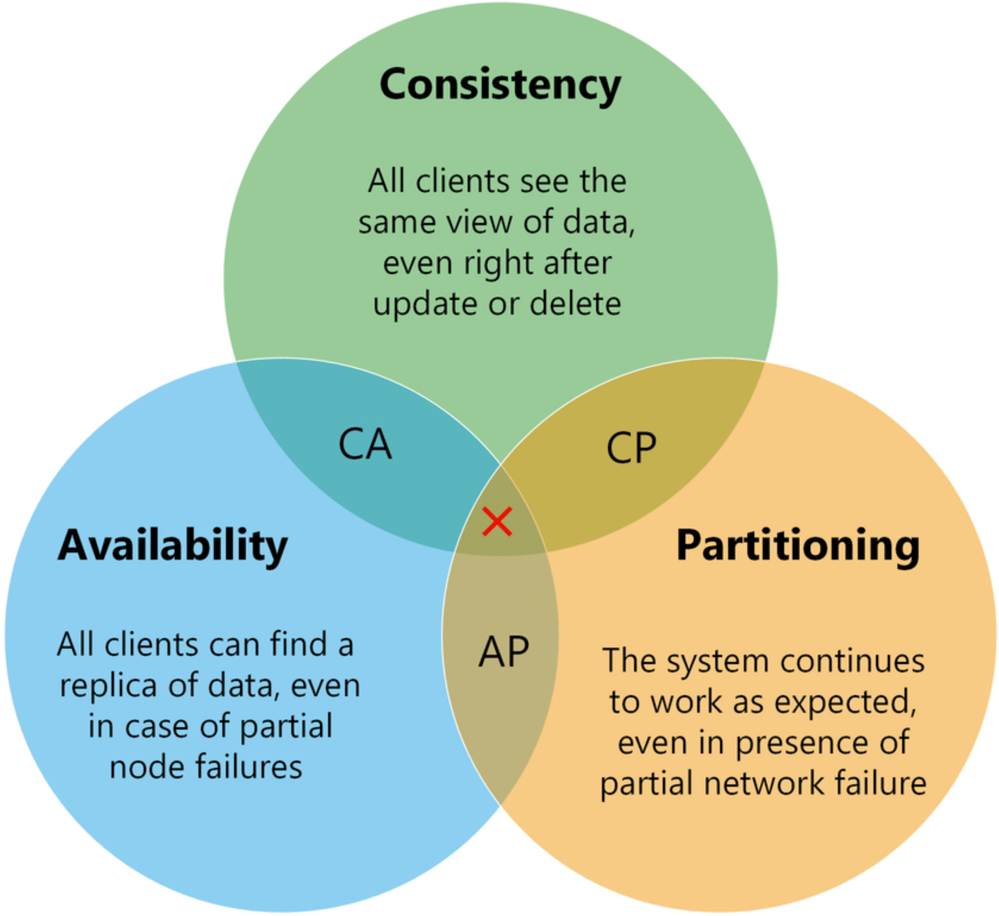
```
{
    "_id": 1,
    "first_name": "Leslie",
    "last_name": "Yepp",
    "cell": "8125552344",
    "city": "Pawnee",
    "hobbies": ["scrapbooking", "eating waffles", "working"]
}
```

NoSQL

The CAP theorem

Also named **Brewer's theorem** after computer scientist Eric Brewer, states that any distributed data store can only provide two of the following three guarantees:

1. **Consistency**
Every read receives the most recent write or an error.
2. **Availability**
Every request receives a (non-error) response, without the guarantee that it contains the most recent write.
3. **Partition tolerance**
The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.



NoSQL vs. RDBMS

NoSQL vs. RDBMS

Attributes	SQL Databases	NoSQL Databases
Data Model	Normalized structured data (rows and columns)	1. Documents 2. Key-value Pairs 3. Rows with dynamic columns 4. Graph
Development History	Developed in the 1970s with a focus on reducing data duplication	Developed in the late 2000s with a focus on scaling and allowing for rapid application change driven by agile and DevOps practices.
Examples	Oracle, MySQL, Microsoft SQL Server, and PostgreSQL	Document: MongoDB and CouchDB, Key-value: Redis and DynamoDB, Wide-column: Cassandra and HBase, Graph: Neo4j and Amazon Neptune
Community support	Closed-sourced with licencing fees	Open-source community

NoSQL vs. RDBMS

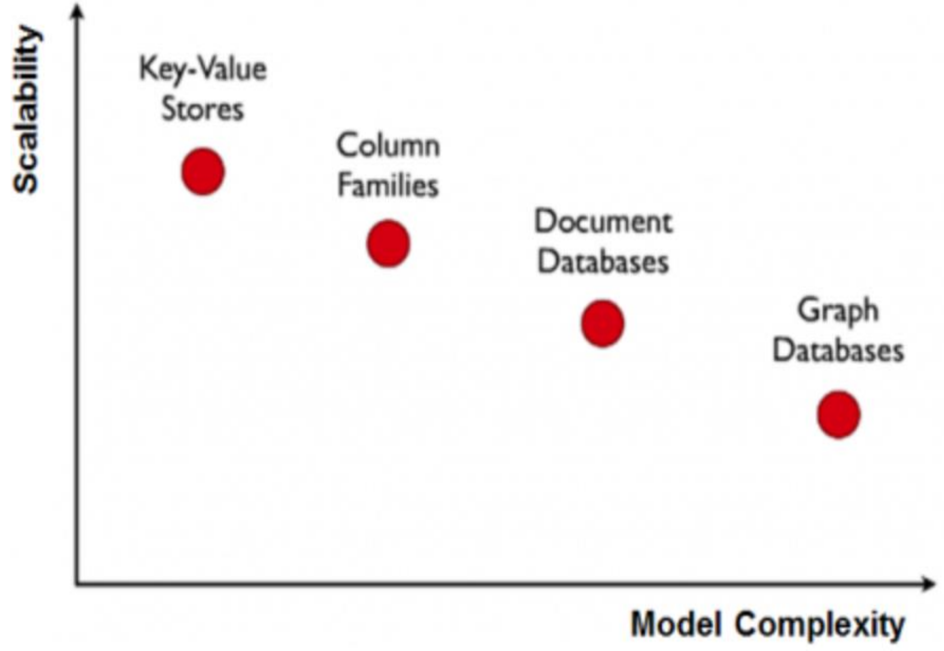
Attributes	SQL Databases	NoSQL Databases
Schema Design	Pre-defined and rigid	Dynamic and Flexible (Schema less)
Scaling	Vertical / Scale-up Increasing hardware capacity, such as faster CPU and larger RAM	Horizontal / Scale-out Adding more servers
Transaction Guarantees	All support ACID (Atomicity, Consistency, Isolation, Durability)	Most support BASE (Basically Available, Soft state, Eventually consistent)
Joins	Typically required	Typically not required
CAP theorem	Not applicable	Eventual consistency and prioritize Availability and Partition tolerance
Query Language	Structured Query Language (SQL)	No declarative query language

Types of NoSQL Databases

Types of NoSQL Databases

NoSQL Databases are mainly categorized into four types:

1. **Key-value stores** (Riak, Redis server …)
2. **Document databases** (MongoDB, RavenDB)
3. **Column oriented databases**(BigTable, Cassandra …)
4. **Graph databases** (Neo4J, InfoGrid …)



Key-value stores

- ❑ **Data Model**
  - Works as a simple hash table where each key is unique (mapping)
  - **Key-value pairs:**
    - Key (id, identifier, primary key, unique)
    - Value: could be binary object, string etc.

Key	Value
Name	Joe Bloggs
Age	42
Occupation	Stunt Double
Height	175cm
Weight	77kg

- ❑ **Query patterns**
  - Create, update or remove value for a **given key**
  - Get value for a **given key**
- ❑ **When to use**
  - It is designed in a way to **handle lots of data** and heavy load
  - When values are only accessed via keys, e.g. shopping carts, session data

Document databases

❑ Data Model

- **Documents:**
  - Document-Oriented NoSQL DB stores and retrieves data in the form documents
  - Self-describing, Hierarchical tree structures (JSON, XML)

❑ Query Pattern

- Create, update or remove a document
- Retrieve documents according to complex query conditions

❑ When to use

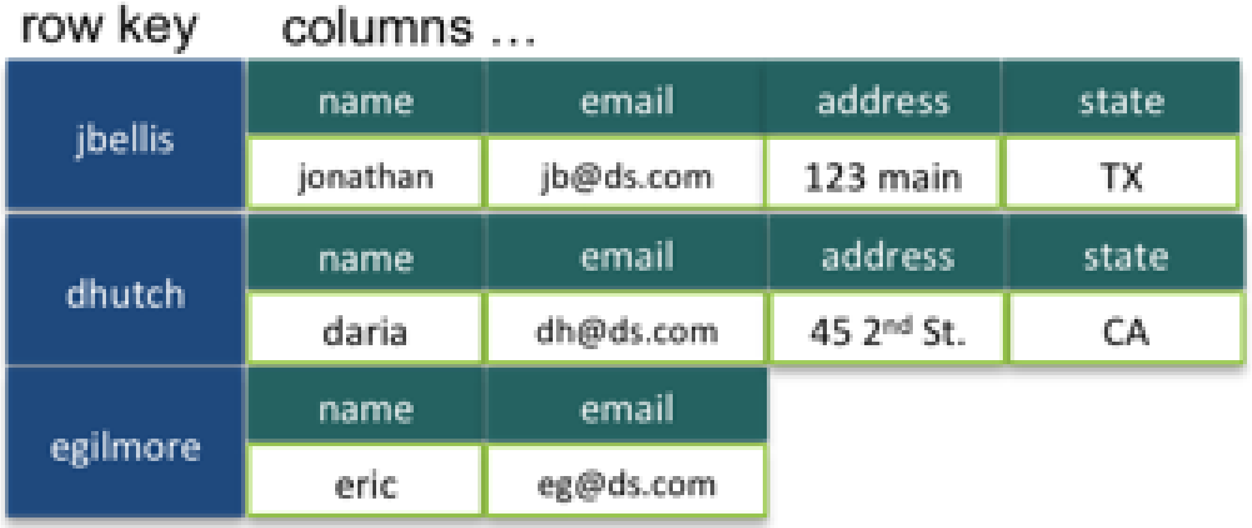
- **Set operations** involving multiple documents

A Database contains a collection, and a collection contains documents and the documents contain data, which are related to each other.

Customer Document


```
"customer" =
{
   "id": "Customer:1",
   "firstName": "John",
   "lastName": "Wick",
   "age: 25,
   "address": {
        "country": "US",
        "city": "New York",
        "state": "NY"
        "street": "21 2nd Street",
    },
   "hobbies": [ Football, Hiking ],
   "phoneNumbers": [
     {
        "type": "Home",
        "number": "212 555-1234"
     },
     {
        "type": "Office",
        "number": "616 565-6789"
     }
   ]
}
```

Column-oriented databases

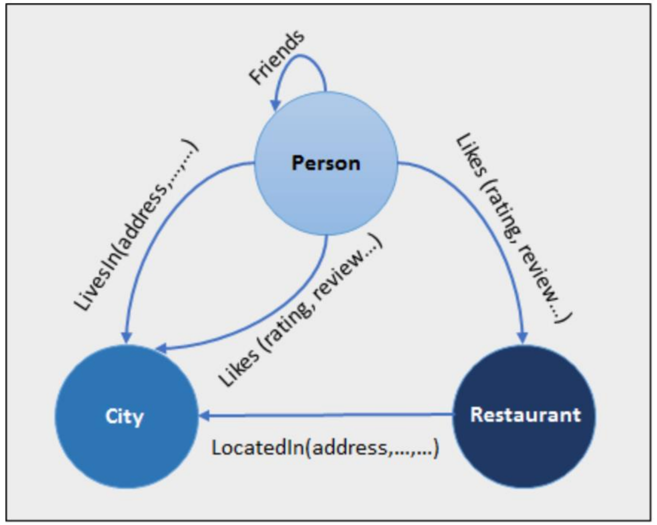


❑ **Data Model**
- **Column family (table)**
  - Table is a collection of similar rows (not necessarily identical)
- **Row**
  - Row is a collection of columns
  - Associated with a unique row key
- **Column**
  - Column consists of a column name and column value

❑ **Query Pattern**
- Create, update or remove a row within a given column family

❑ **When to use**
- for structured flat data with similar schema, e.g. blogs

Graph databases



❑ Data Model

- **Property**
  - Directed / undirected graphs
  - Collections of: nodes(entities) and edges(relationships)
- Both the nodes and relationships can be associated with additional properties
- Every node and edge has a unique identifier

❑ Query Pattern

- Create, update or remove a node / edge in a graph
- Graph algorithms (graph traversals, shortest paths, spanning trees, …)

❑ When to use

- For graph structure, e.g. social networking, recommendation engine …

XML, JSON Data Formats

Document oriented database

XML – Mark-up Language

- XML was designed to store and transport data.
- XML is an extensible markup language like HTML. XML allows you to create your own self-descriptive tags, or language, that suits your application.
- XML was designed to carry data, not to display that data.
- Mark-up code of XML is easy to understand for a human.
- The structured format is easy to read and write from programs.
- It is case sensitive.

The Difference Between XML and HTML

XML and HTML were designed with different goals:

- XML was designed to carry data - with focus on what data is.
- HTML was designed to display data - with focus on how data looks.
- XML tags are not predefined like HTML tags are.
- White spaces are preserved in XML and not in HTML.

XML

```
<note>
  <to>John</to>
  <from>Eva</from>
  <heading>Reminder</heading>
  <body>Don't forget meeting this weekend!</body>
</note>
```

HTML

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example</title>
    </head>
    <body>
        <p>A simple HTML page with one paragraph.</p>
    </body>
</html
```

Extensible Markup Language (XML)

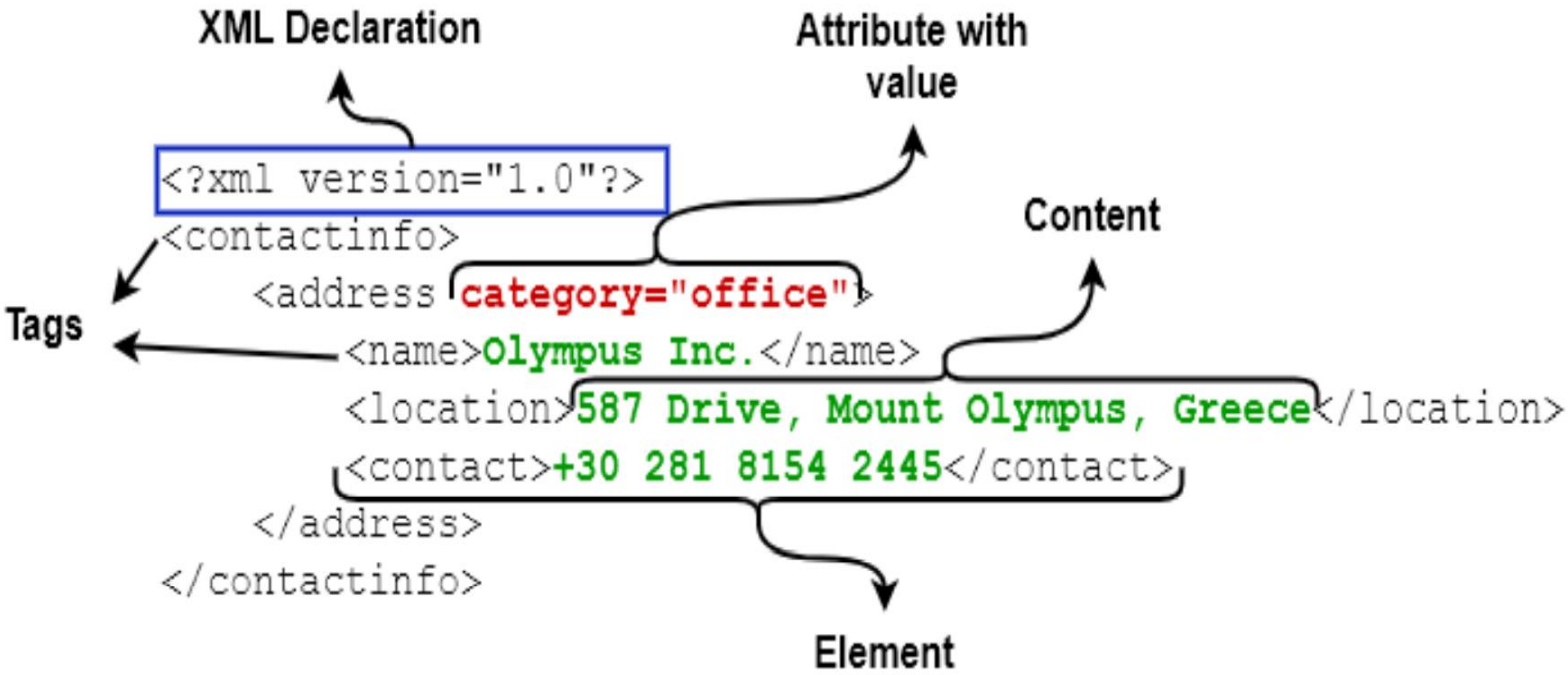
```xml
<?xml version="1.0"?>
<contactinfo>
    <address category="office">
        <name>Olympus Inc.</name>
        <location>587 Drive, Mount Olympus, Greece</location>
        <contact>+30 281 8154 2445</contact>
    </address>
</contactinfo>
```

Extensible

```xml
<?xml version="1.0"?>
<contactinfo>
    <name>John Doe</name>
    <address category="office">
        <name>Olympus Inc.</name>
        <location>587 Drive, Mount Olympus, Greece</location>
        <contact>+30 281 8154 2445</contact>
    </address>
    <address category="home">
        <street>54 Moon Beam Drive</street>
        <houseno>8</houseno>
        <postalcode>487510</postalcode>
    </address>
</contactinfo>
```

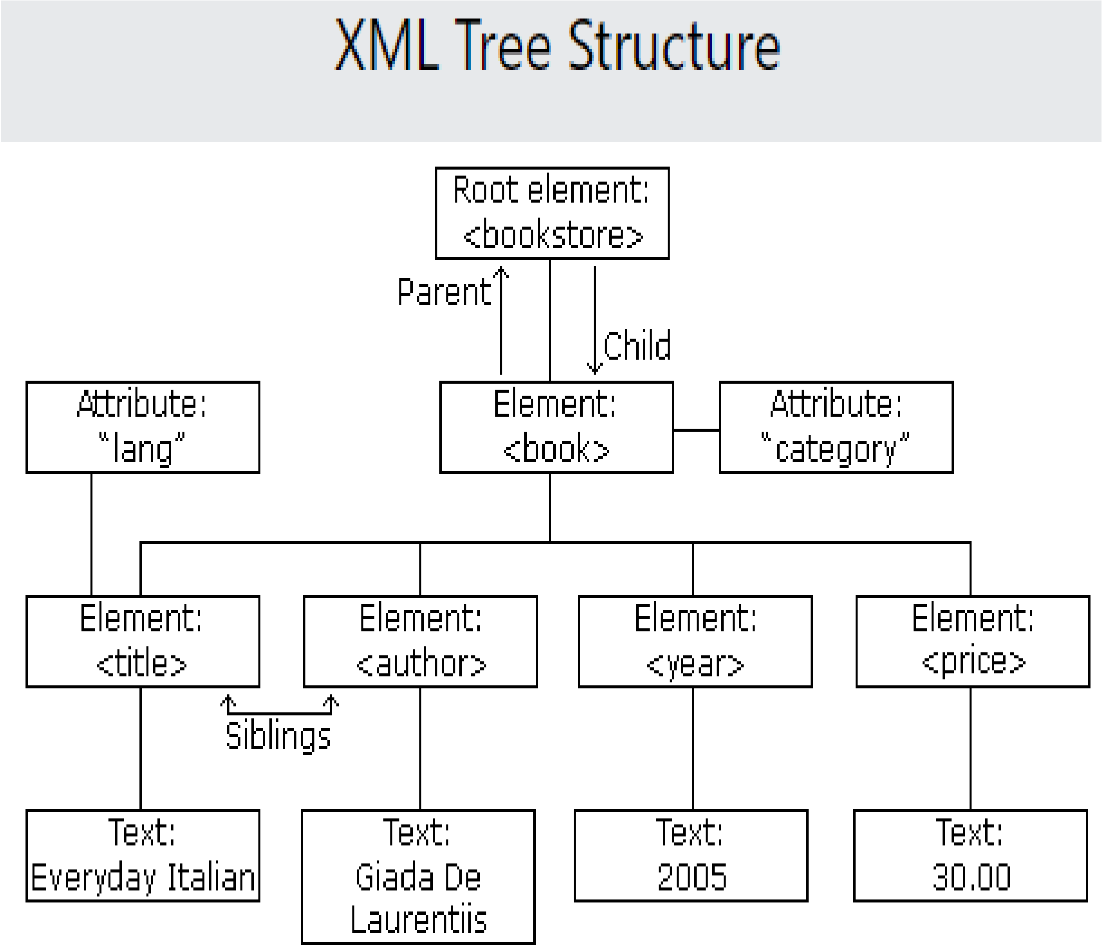
More information pertaining to **contactinfo** can be added by adding more descriptive tags to **contactinfo** tag.

Structure of XML



Structure of XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

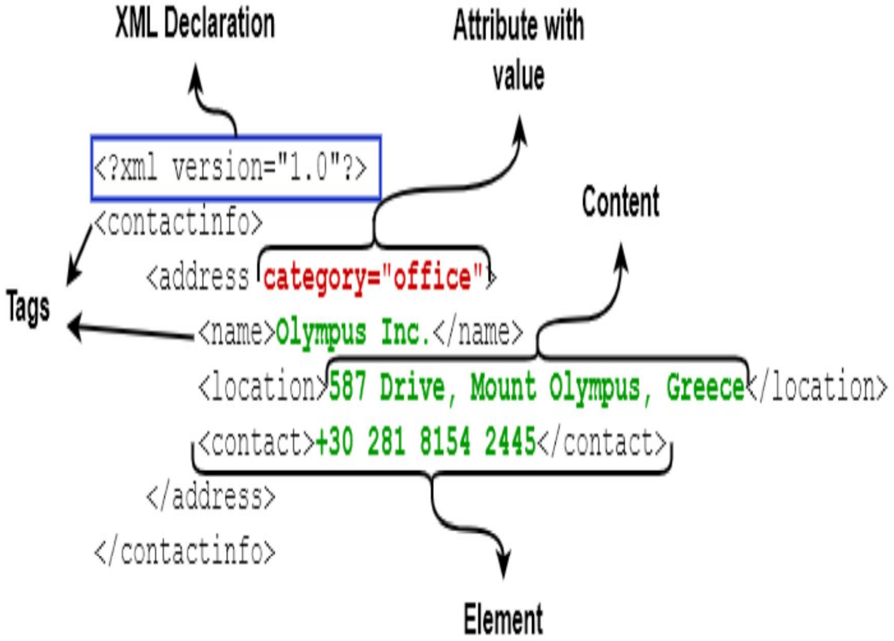


- **XML declaration**
  - This is the declaration at the top of every XML document that describes the XML document.

- **Unicode characters**
  - XML documents uses strings of Unicode characters but not all Unicode characters are valid.

- **Markup and Content**
  - Information making up the XML document are split into 2 types
    - Markup strings included in <>.
    - Content are the strings within the markup.
- **Tags**
  - Tags is the markup string that begins with < and ends with >.
  - There are 3 types of tags
    - start-tag <name>, end-tag </name>
    - empty-element tag <extension />

```
<phone>
    <entry>
        <name>Chip</name>
        <extension number="3"/>
    </entry>
</phone>
```



- **Elements**
  - Elements are a matching pair of tags and/or content that begins with a start-tag and ends with a matching end-tag, or consist of only of the empty-element tag.

- **Attributes**
  - Attributes are the name-value pairs that exists within the start-tag or empty-element tag.

JavaScript Object Notation (JSON)

- JSON
  - It is an open-standard data transition format to transfer the data from one system to another.
  - It can transfer data between two computers, database, programs etc.
  - It is text-based and can be easily read by humans and machines.
  - It is commonly used for transmitting data in web applications.
  - It is platform independent.
  - Its extension is **.json**.

- Use **key-value pairs** to store information

```json
{
    "brand": "Sketchers",
    "colour": "black",
    "size": 38,
    "insole": "memory foam"
}
```

JSON Syntax

- JSON defines only two data structures: **objects** and **arrays**.
- An **object** is a set of **name-value pairs**, and an **array** is a **list of values**.
- JSON defines **seven value types**: string, number, object, array, true, false, and null.

```json
{
    "customer": {
        "name": "John Doe",
        "weightInKg": 70,
        "head": {
            "hair": {
                "hairColour": "brunette",
                "length": "short",
                "style": "crew-cut"
            },
            "eyeColour": "blue",
            "piercings": null,
        },
        "tatoos": ["dove", "eagle", "crane"],
        "isMarried": false
    }
}
```

JSON has the following syntax.

1. Objects are enclosed in **braces ({})**, their name-value pairs are separated by a **comma (,)** and the name and value in a pair are separated by a **colon (:)**. Names in an object are **strings**, whereas values may be of any of the **seven value types**, including another object or an array.

2. Arrays are enclosed in **brackets ([])**, and their values are separated by a comma (,). Each value in an array may be of a different type, including another array or an object.

3. When objects and arrays contain other objects or arrays, the data has a **tree-like structure**.

JSON Syntax

```
{
  "firstName": "Duke",
  "lastName": "Java",
  "age": 18,
  "streetAddress": "100 Internet Dr",
  "city": "JavaTown",
  "state": "JA",
  "postalCode": "12345",
  "phoneNumbers": [
    { "Mobile": "111-111-1111" },
    { "Home": "222-222-2222" }
  ]
}
```

The value for the name *"phoneNumbers"* is an array whose elements are two objects.

JSON object tree structure

```
{
    "customer": {
        "name": "John Doe",
        "weightInKg": 70,
        "head": {
            "hair": {
                "hairColour": "brunette",
                "length": "short",
                "style": "crew-cut"
            },
            "eyeColour": "blue",
            "piercings": null,
        },
        "tatoos": ["dove", "eagle", "crane"],
        "isMarried": false
    }
}
```

Datatype in JSON

```
{
    "customer": {
        "name": "John Doe",
        "weightInKg": 70,
        "head": {
            "hair": {
                "hairColour": "brunette",
                "length": "short",
                "style": "crew-cut"
            },
            "eyeColour": "blue",
            "piercings": null,
        },
        "tatoos": ["dove", "eagle", "crane"],
        "isMarried": false
    }
}
```

- **Object** (root of a tree/subtree)
  - customer, head, hair are of type Object
- **String**
  - name, hairColour have value of type string
- **Number**
  - weightInKg has value of type number
- **Boolean**
  - isMarried has value of type boolean
- **Null**
  - piercings
- **Array**
  - tatoos

XML vs. JSON

JSON	XML
Its files are very easy to read as compared to XML	Its documents are comparatively difficult to read and interpret
JSON types: string, number, array, Boolean	All XML data should be string
It supports array	It doesn’t supports array
It doesn’t use end tag	It has start and end tags
It is less secured	It is more secured than JSON
It doesn’t supports comments	It supports comments
It supports only UTF-8 encoding	It supports various encoding
JSON is supported by most browsers	Cross-browser XML parsing can be tricky
JSON has no display capabilities	XML offers the capability to display data because it is a markup language
JSON supports only text and number data type	XML support various data types such as number, text, images, charts, graphs, etc
Data is readily accessible as JSON objects	XML data needs to be parsed

JSON code vs. XML code

XML

```
<empinfo>
    <employees>
        <employee>
            <name>James Kirk</name>
            <age>40></age>
        </employee>
        <employee>
            <name>Jean-Luc Picard</name>
            <age>45</age>
        </employee>
        <employee>
            <name>Wesley Crusher</name>
            <age>27</age>
        </employee>
    </employees>
</empinfo>
```

JSON

```
{   "empinfo" :
        {
                "employees" :  [
                {
                    "name" : "James Kirk",
                    "age" : 40,
                },
                {
                    "name" : "Jean-Luc Picard",
                    "age" : 45,
                },
                {
                    "name" : "Wesley Crusher",
                    "age" : 27,
                }
                                ]
        }
    }
```