Presented by

Dr. Trupti Padiya

School of Computing and Creative Technologies

Advanced Databases UFCFU3-15-3

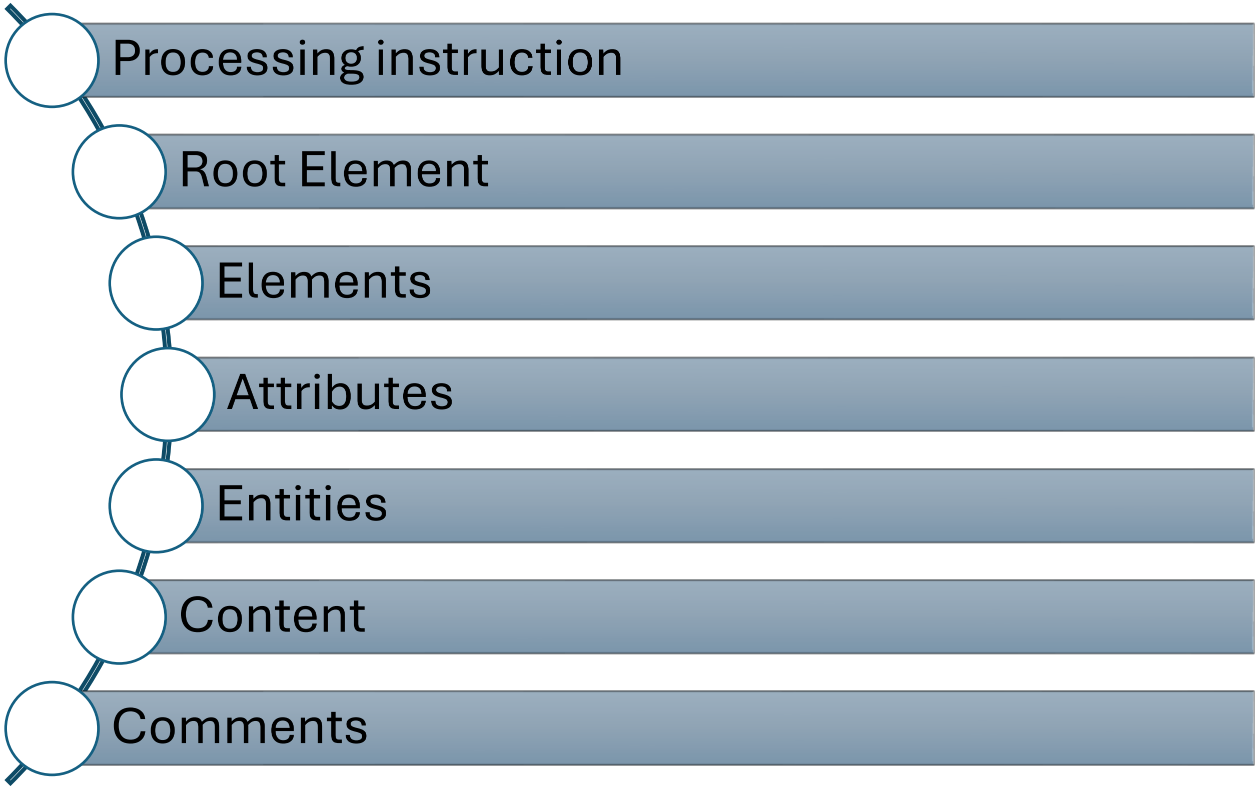
NoSQL Databases: XML, JSON, MongoDB

Slides with modification adapted from Dr. Elias Pimenidis

XML

- XML(eXtensive Markup Language)
- Is a self-describing (schema-less) language
  - Machine readable
  - Human readable
- Embedded descriptive tags – Similar to HTML in a way
  - Tags are not pre-defined, to use it, you need to define your own tags
- XML is designed for **data availability, sharing & transport**
  - Is recommended by W3C for information exchange over the Internet
- Is a software and hardware independent language
  - Enables programmers to store data in a well-formed structure
- Transfers data between various heterogeneous systems over the network

XML Components



- Processing instruction
- Root Element
- Elements
- Attributes
- Entities
- Content
- Comments

XML components

- **Processing Instruction (PI)**
  - Specifies how an XML document is processed
  - `<?xml version="1.0" encoding="UTF-8"?>`
- **Tags**
  - Specify the name of information presented in an XML document
  - Must have opening (<>) and closing (</>) brackets that encloses the name of the tag (well-formed)
  - `<first_name>Allen</first_name>`

XML Components

- **Root Element**
  - Every XML document must have one **root element**
- **Elements**
  - Are the basic building blocks of an XML document
  - Are represented with the help of tags
  - Are used to describe data in an XML document
  - Can contain one or more elements
  - Elements that contain child elements are known as root elements

```
<name>
    <first_name>Allen</first_name>
    <last_name>Thomas</last_name>
</name>
```

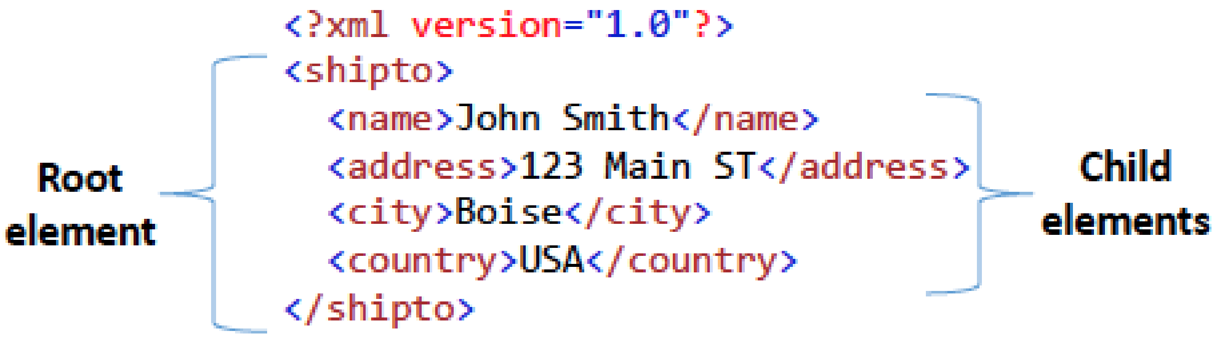
XML Components

- **Content**
  - Refers to the information represented by the elements
  - Can be categorized into following types:
    - Character Content
    - Element Content
    - Mixed Content
- **Attributes**
  - Allow programmers to provide additional information about the elements
  - Are created in the form of name-value pairs as part of an opening tag

```
<emp emp_id="001”>
    <first_name>Allen</first_name>
    <last_name>Thomas</last_name>
</emp>
```

Well Formed XML

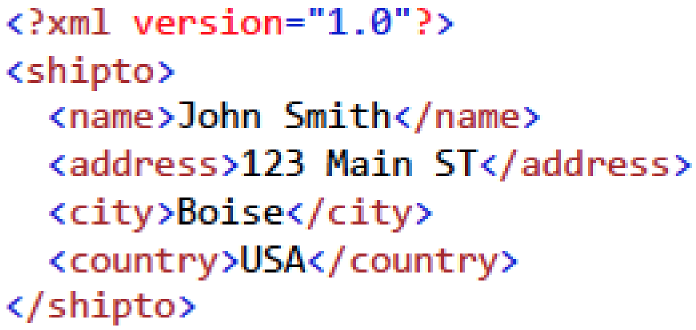
- XML uses a hierarchical, nested tree structure
- An XML tree starts at a root element and branches from the root to child elements.
- Elements can have sub elements (child elements)



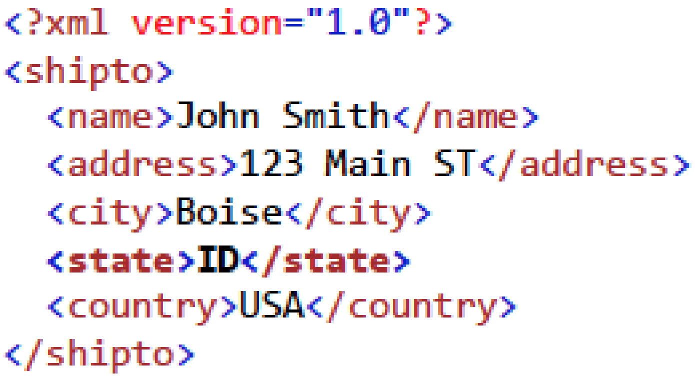
```
<?xml version="1.0"?>
<shipto>
  <name>John Smith</name>
  <address>123 Main ST</address>
  <city>Boise</city>
  <country>USA</country>
</shipto>
```

XML is Extensible

- XML is extensible, in that elements can be easily added as needed
- If the <state> element is added below, older applications using the original version will still work



```xml
<?xml version="1.0"?>
<shipto>
  <name>John Smith</name>
  <address>123 Main ST</address>
  <city>Boise</city>
  <country>USA</country>
</shipto>
```

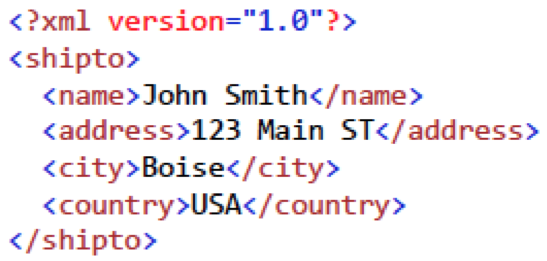


```xml
<?xml version="1.0"?>
<shipto>
  <name>John Smith</name>
  <address>123 Main ST</address>
  <city>Boise</city>
  <state>ID</state>
  <country>USA</country>
</shipto>
```

XML is Self Describing

- XML is self-describing (sort of) with the use of element tags
- Machine/human-readable format
- Tags describe the content of the element (sort of)

From reading the tags, it's pretty clear that we're talking about a "Ship To" address that contains the name, address, city & country.



```
<?xml version="1.0"?>
<shipto>
  <name>John Smith</name>
  <address>123 Main ST</address>
  <city>Boise</city>
  <country>USA</country>
</shipto>
```

But it doesn't provide full metadata, e.g.:

- What's the data type?
- What's the business definition?
- Is <name> a required field?

XML Metadata – XML Schema

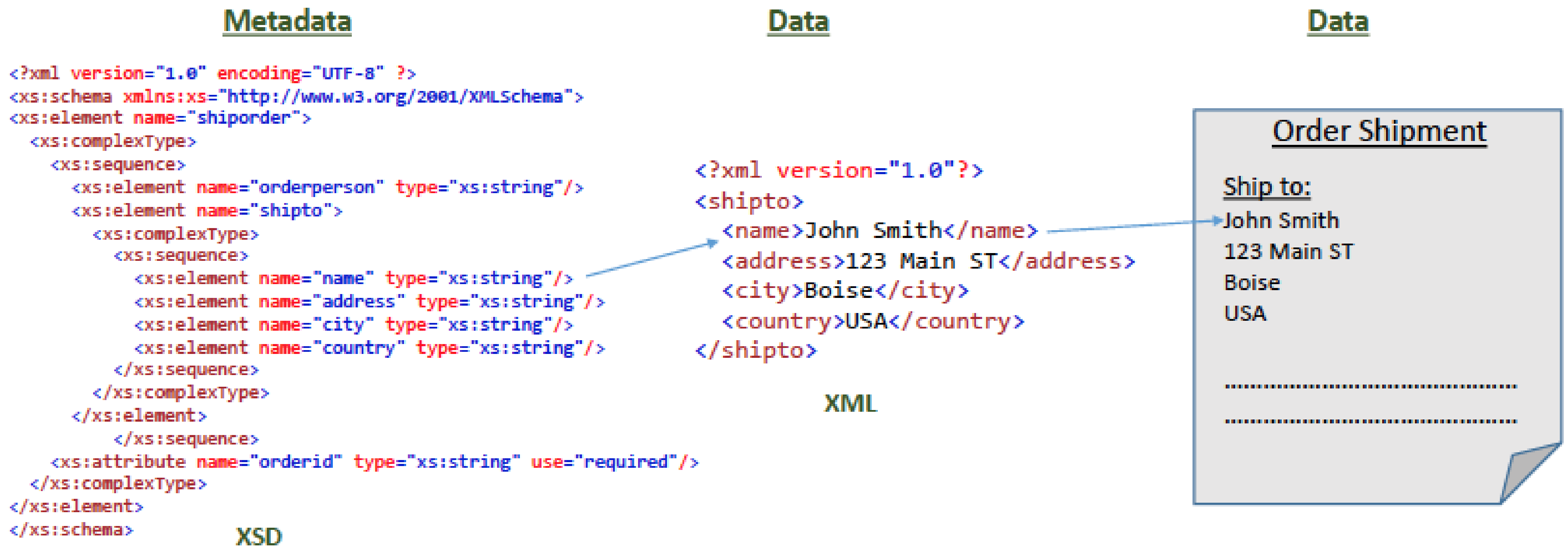
- XML schemas follow a universal standard for data communication over the Internet safe
- For example, a date like: "01-07-2015" can be interpreted as 7 January in some countries, whereas in other countries as 1 July. However, an XML element accepts date in the fixed format "YYYYMM- DD" to ensure its correct interpretation
- XML schemas support data types that allow programmers to:
  - Specify the acceptable content in the document
  - Ensure the validity of data
  - Specify data formats
  - Work with databases
  - Apply restrictions on data
  - Convert data from one data type to another

Namespace

- Namespace
  - A namespace can be considered as a virtual space that is identified by a Uniform Resource Identifier (URI)
  - It is represented in the form of a string that uniquely identifies the elements and attributes from different schemas (e.g. **xmlns**:prefix="URI”)
  - namespaces are used to avoid conflict between the elements having the same name by using a name prefix

XML Metadata Schema

- Similar to DDL, an XML Schema (XSD) defines the structure of an XML document



Metadata

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="shiporder">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="orderperson" type="xs:string"/>
      <xs:element name="shipto">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
          </xs:sequence>
    <xs:attribute name="orderid" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
</xs:schema>
```

XSD

Data

```
<?xml version="1.0"?>
<shipto>
  <name>John Smith</name>
  <address>123 Main ST</address>
  <city>Boise</city>
  <country>USA</country>
</shipto>
```

XML

Data

Order Shipment

Ship to:
John Smith
123 Main ST
Boise
USA

…………………………………………

…………………………………………

XML Metadata – Document Type Definition

- XML DTD
- DTD - Stands for Document Type Definition
  - Alternative to XML Schema
  - Is used to define the structure and the valid elements and attributes of an XML document
  - Verifies the data in an XML document.
  - Can be used to verify data of a local file or data received over the network

```
<?xml version="1.0"?>
<!DOCTYPE name [
<!ELEMENT name (first_name, last_name)>
<!ELEMENT first_name (#PCDATA)>
<!ELEMENT last_name (#PCDATA)>
]>
```

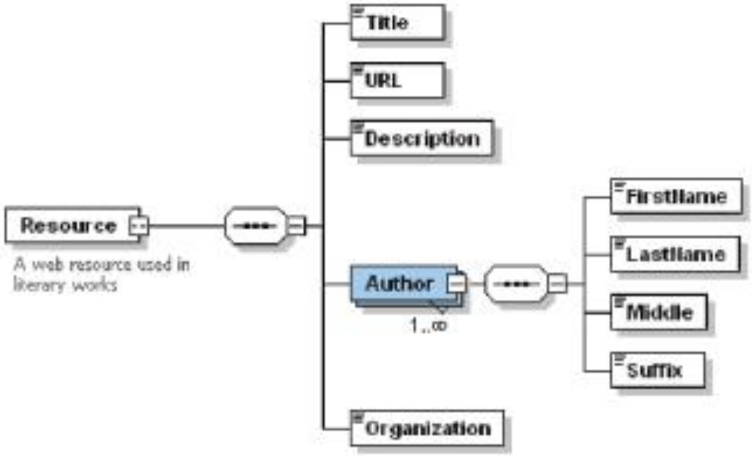
XSD vs DTD

XSD	DTD
XSD stands for XML Schema Definition.	DTD stands for **Document Type Definition.**
XSDs are written in XML.	DTDs are derived from **SGML** syntax.
XSD **supports datatypes** for elements and attributes.	DTD **doesn't support datatypes**.
XSD **supports namespace**.	DTD **doesn't support namespace**.
XSD **defines order** for child elements.	DTD **doesn't define order** for child elements.
XSD is **extensible**.	DTD is **not extensible**.
XSD is **simple to learn** because you don't need to learn new language.	DTD is **not simple to learn**.
XSD provides **more control** on XML structure.	DTD provides **less control** on XML structure.

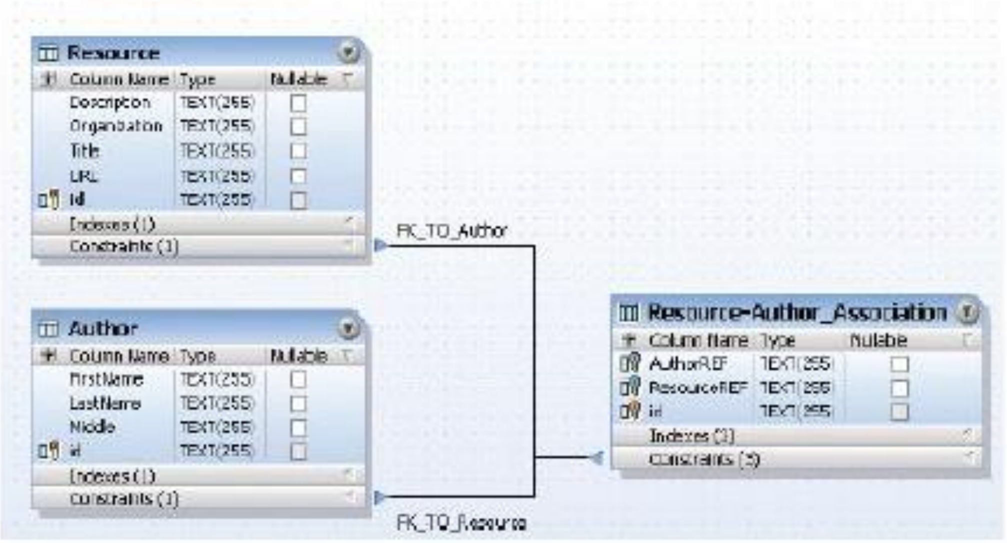
Integrating XML with Relational Databases

- XML can be translated into relational databases, and vice-versa

XML Model Diagram



Relational Model Diagram



Resource

Column Name	Type	Nullable
Description	TEXT(255)	□
Organization	TEXT(255)	□
Title	TEXT(255)	□
URL	TEXT(255)	□
id	TEXT(255)	□

Indexes (1)
Constraints (1)

Author

Column Name	Type	Nullable
FirstName	TEXT(255)	□
LastName	TEXT(255)	□
Middle	TEXT(255)	□
id	TEXT(255)	□

Indexes (1)
Constraints (1)

Resource-Author_Association

Column Name	Type	Nullable
AuthorREF	TEXT(255)	□
ResourceREF	TEXT(255)	□
id	TEXT(255)	□

Indexes (3)
Constraints (3)

XML Benefits

- **Data Interchange:**
  - It allows programmers to store data in textual format that can be used as a standard to interchange data
- **Data Transformation:**
  - As per requirements, data can be stored in the form of text, object, or data in a database. The stored data can be extracted by the client application in the required format
- **Smart Searches:**
  - It allows programmers to specify whether they want to search information based on text or tags, and returns the information that matches the search criteria
- **Fast Updates:**
  - Use of XML allows fast update of information, as only text needs to be updated
- **CSS and XSL Support:**
  - XML supports CSS (Cascading Style Sheet) and XSL (Extensible Style Sheet) languages that can be used to apply required formatting of an XML document

JSON

- **JavaScript Object Notation** is a minimal, readable format for structuring data. It is used primarily to transmit data between a server and web application, as an alternative to XML
- Is an open standard light-weight format that is used to store and exchange data
- Is language independent format that uses human readable text to transmit data objects
- Syntactically, JSON is similar to the code for creating JavaScript objects

JSON Object

- Unordered sets of name/value pairs
  - Begins with **{ (left brace)**
  - Ends with **} (right brace)**
  - Each name is followed by **: (colon)**
  - Name/value pairs are separated by **, (comma)**

```
var employeeData = {
  "employee_id": 1234567,
  "name": "Jeff Fox",
  "hire_date": "1/1/2013",
  "location": "Norwalk, CT",
  "consultant": false
};
```

JSON Values

- A JSON object can include the following types of values:
  - A numeric value
  - A string
  - A Boolean value
  - An array
  - An object
  - A null value
- `"fName": "Jane" \\Storing string value`
- `"lName": "Doe" \\Storing string value`
- `"isAlive": true \\Storing Boolean value`
- `"age": 23 \\Storing integer value`
- `"children": [] \\Storing an array`
- `"spouse": null \\Storing null`

JSON Array

- JSON arrays can be created by using square brackets, as shown in the following code snippet:

```
{
"fName": "Jane",
"lName": "Doe",
"isAlive": true,
"age": 23,
"ContactNumber":[
    {"type":"Mobile", "Number":"+9198765" }
    {"type":"Office", "Number":"+9124456" }],
"children": [],
"spouse": null

}
```

- Since JSON uses the same syntax as that of JavaScript objects, JSON arrays can be accessed in the same way as in JavaScript

JSON vs XML

- **Similarities**
- Both are human-readable, that is, self-describing
- Both represent hierarchical structure, that is, values within values
- Both can be accessed and parsed by almost every programming language
- Both can be accessed and fetched with an XMLHttpRequest object

- **Dissimilarities**
- XML needs an XML parser, whereas, a standard JavaScript method can be used to parse JSON
- There is no need of end tag in JSON
- JSON is much shorter as compared to XML
- It is easy to read and write JSON
- JSON can be used with arrays

JSON Syntax vs XML Syntax

- The following code snippet depicts a JSON example that defines a student

```
{"students":[
{"fName":"Jenny", "lName":"Watson"},
{"fName":"Dean", "lName":"Smith"}
]}
```

- The following code snippet depicts an XML example that defines a student:

```
<students>
      <student>
            <fName>Jenny</fName>
            <lName>Watson</lName>
      </student>
      <student>
            <fName>Dean</fName>
            <lName>Smith</lName>
      </student>

</students>
```

Reading Data from JSON

- A most common usage of JSON objects is to read/fetch data from a Web server in JSON format, and display it on an HTML Web page.
- To read data from a JSON object, you can use the JSON.parse()method provided by JavaScript.
- The syntax for JSON.parse() method is as follows:

```
var obj = JSON.parse(text);
```

- The following code snippet depicts how to use the JSON.parse()method:

```
var jsonData =
'{"fName": "Jane", "lName": "Doe", "isAlive": true,
"age": 23}';
var contact = JSON.parse(jsonData);
document.write(contact.lName+", "+contact.fName);
```

JSON Data on Webpage

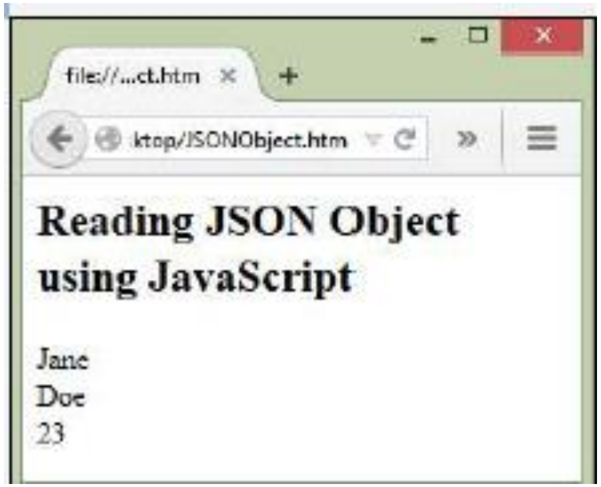
```
<html>
<body>
<h2>Reading JSON Object using JavaScript</h2>
<p id="pData"></p>
<script>

var jsonData ='{"fName": "Jane", "lName": "Doe", "isAlive":
true, "age": 23}’;

var contact = JSON.parse(jsonData);

document.getElementById("pData").

innerHTML = contact.fName + "<br>" +
           contact.lName + "<br>" +
           contact.age;
</script>
</body>
</html>
```



Reading JSON Object using JavaScript

Jane
Doe
23

JSON - Advantages

- Easy to understand, manipulate and generate
- Completely language independent
- Straightforward syntax
- Based on a subset of the JavaScript Programming Language
  - Can be natively converted in JavaScript using `eval()`
- Supported by all major JavaScript frameworks
- Supported by most backend technologies
- Plain text formats
- Self-describing (human readable)
- Hierarchical (Values can contain lists of objects or values)

Integrating JSON with Document Store

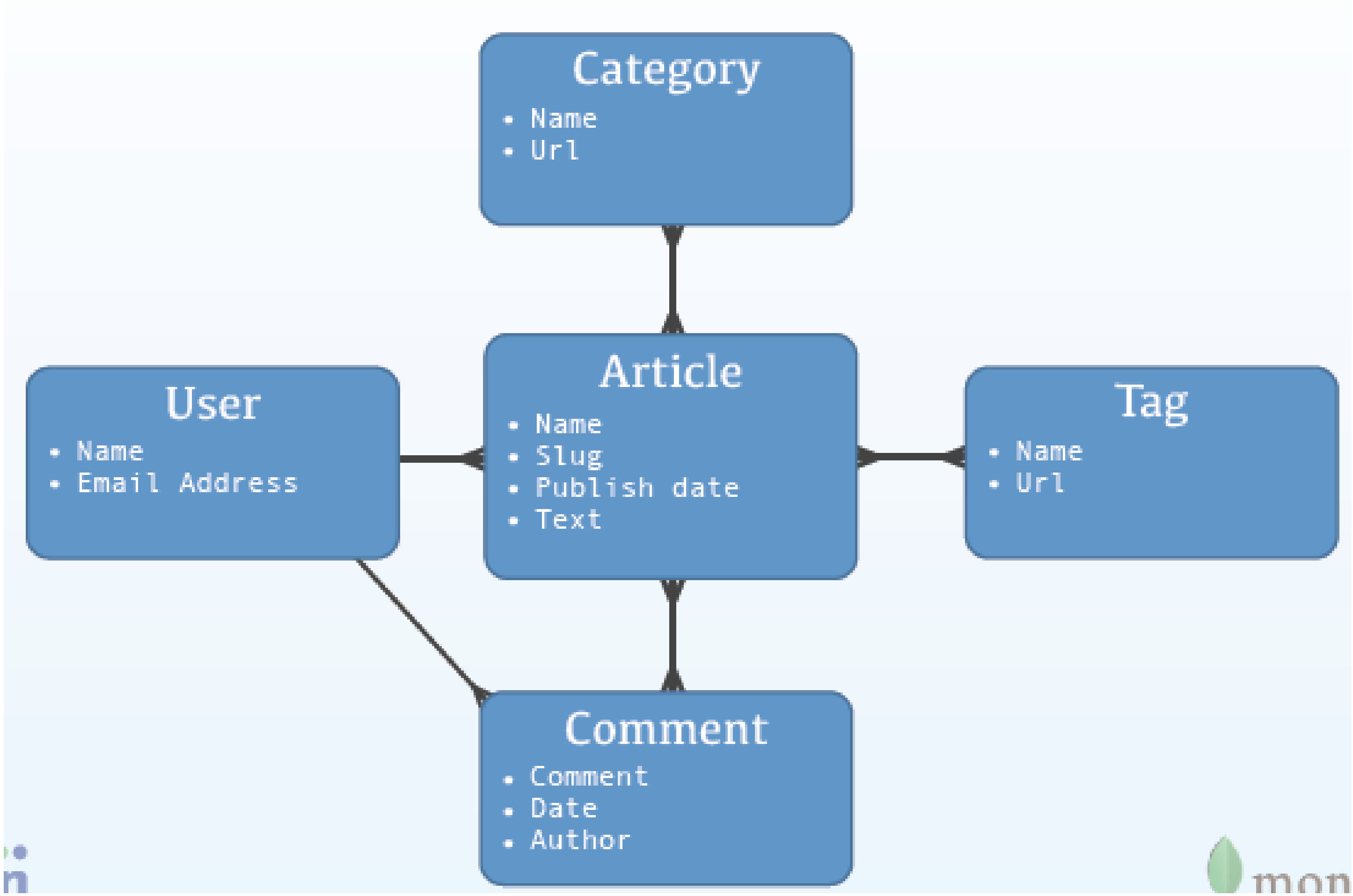
- JSON is often used with document databases, such as **MongoDB**, which uses JSON documents in order to store records
- Document databases are popular ways to store semi-structured information in a flexible way (e.g. multimedia, social media posts, etc.)
- Each Collection can contain numerous documents which could all contain different fields

```
{type: “Artifact”,
medium: “Ceramic”
country: “China”,
}
```

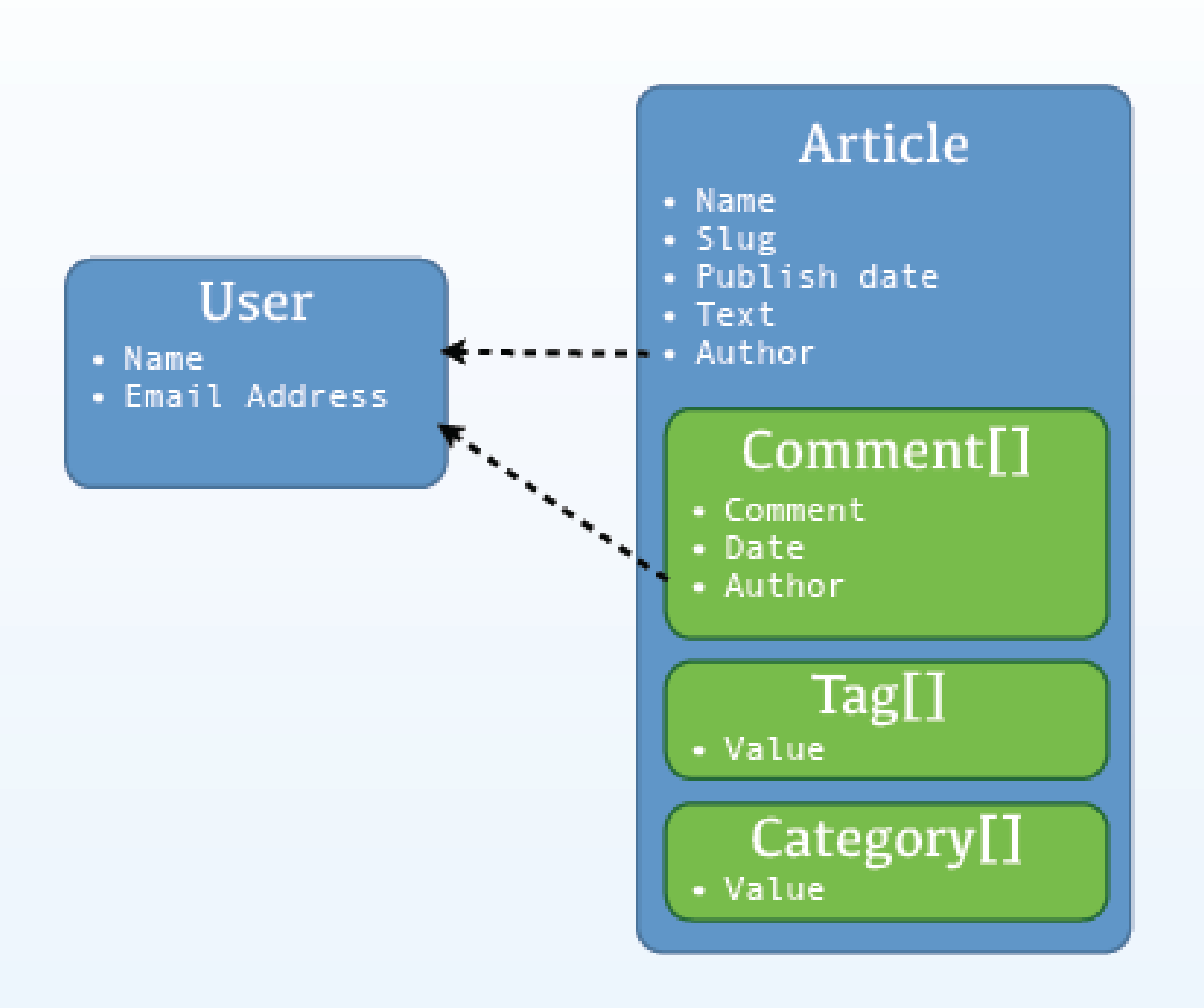
MongoDB Philosophy

- Keep functionality when we can (key/value stores are great, but we need more)
- Non-relational (no joins) makes scaling horizontally practical
- Document data models are good
- Database technology should run anywhere virtualized, cloud, metal, etc.

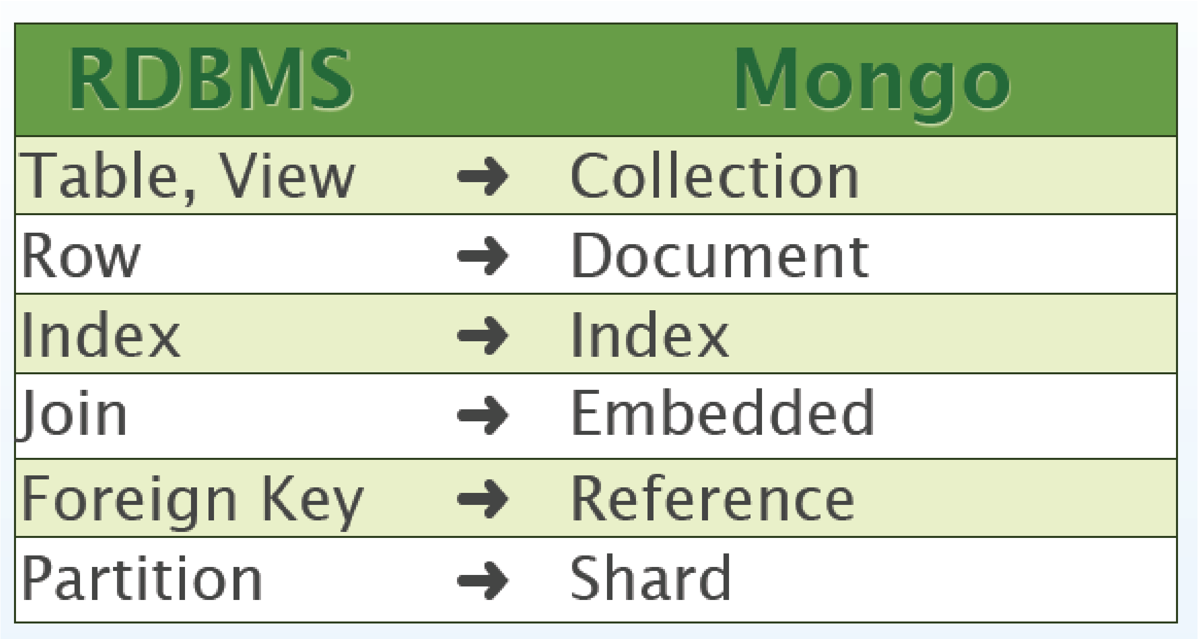
RDMBS and normalized data



Document Store and normalized data



Terminologies



RDBMS		Mongo
Table, View	→	Collection
Row	→	Document
Index	→	Index
Join	→	Embedded
Foreign Key	→	Reference
Partition	→	Shard

MongoDB Commands

- **Select** a database: use database_name
- **Show** databases: show dbs
- **Drop** current database: db.dropDatabase()
- **Get** current database name: db.getName()
- **Show all** collections of current db: show collections
- **Drop** collection: db.collection.drop()
- **Show all indexes** on collection: db.collection.getIndexes()

Basic MongoDB administrative Commands

- **Dump** data from MongoDB:
  - mongodump --db database –c collection_name path_to_dump
- **Restore** data to MongoDB:
  - mongorestore –-db database –c collection_name path_of_dump_file
- **Export** data from MongoDB:
  - mongoexport –-db database –c collection –-out file_name.json
- **Import** data in MongoDB:
  - mongoimport –-db database –c collection –-file file_name.json

Data creation in MongoDB

- How:
  - db.collection_name.insertOne()
  - db.collection_name.insertMany()
- Creates a collection if it does not exist
- Inserts document into the collection
  - _id Field
  - Inserts automatically a unique ObjectId, if no unique id specified in the document
  - Is unique within the collection
    - Comprises of the current time + identifier of the machine + process ID + process specific local counter
- Examples:

```
db.products.insertOne({ “item”: “card”, qty : 15})
//Inserting with id
db.products.insertOne({ _id: 10, “item”: "box", qty: 20 })
```

Removing Data in MongoDB

- How:
  - db.collection_name.deleteOne()
  - db.collection_name.deleteMany()
  - db.orders.deleteMany({ "client" : “Lime Traders" });

Updating data in MongoDB

- How:
  - db.collection_name.updateOne()
  - db.collection_name.updateMany()
  - db.student.updateOne({name: “Tom"}, {$set:{age: 20}})
  - db.student.updateMany({age:18},{$set:{eligible:"true"}})

Reading data in MongoDB

- How: db.collection_name.find()
- **db.Collection_name.find()**
- **selection_criteria (document)**
- **projection (document)**

- db.student.find()
- db.student.find({age:18})
- db.student.find({score:{m1: 67, m2: 79}})
- db.student.find({}, { name: 1, score: 0 })
- db.student.find().sort({ age: 1 })

Reading data in MongoDB

- Comparison operators:
  - $gt, $gte, $lt, $lte, $ne
  - db.scores.find({score : { $gt : 60}}) //greater than
  - db.scores.find({score : { $gte : 60}}) //greater than equal to
  - db.scores.find({score : { $lt : 60}}) //less than
  - db.scores.find({score : { $lte : 60}}) //less than equal to
  - db.scores.find({score : { $ne : 60}}) //not equal to

Reading data in MongoDB

- Logical Operators
- AND: db.scores.find({ “type” : “exam”, score : {$gt : 60}})
  - db.scores.find({ $and : [{ “type” : “exam” },{ score :$gt : 60}]})
- OR: db.scores.find({ $or : [{ “type” : “exam” },{ type : “essay” }]})
- NOT: db.scores.find({ score : {$not : { $gt : 60}}})

Reading data in MongoDB

- Finding inside arrays

```
{ _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] }
{ _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] }
{ _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 4 ] }
```

- db.inventory.find({ratings : [5,8,9]}) //exact match on the array
- db.inventory.find({ ratings: 5 }) //find where array contains 5
- db.inventory.find({ 'ratings.0': 5 }) //find where ratings array contains 5 as the 1st element
- db.inventory.find({ratings : {$all :[5, 8]}}) //find where ratings array contains 5 and 8 both as necessary elements.
- db.inventory.find({ratings : {$in :[9, 4]}}) //find where ratings array contains 5 or 8

Reading data in MongoDB

- How: db.collection_name.find(query)
- Accessing elements using DOT notation

```
    {
    "_id" :
ObjectId("540cb0e5ff0918e673f1cce9"),
    "name" : "Howard Hughes",
    "email" : {
        "work" : "hhughes@hughes.com",
        "personal" : hhughes@gmail.com
        }
    }
```

- db.users.find({"email.work":"hhughes@hughes.com"})

Data aggregation in MongoDB

- Aggregation refers to performing operations in order to produce aggregated results. i.e. similar to Group by in SQL, count(), Avg(), Sum() etc
- Aggregation is generally done through **aggregation pipeline** and other methods like Map-Reduce
- In this module we will use aggregation pipeline

Aggregation pipeline

- Multiple stages
- Stages in pipeline may include operations like: $match, $group, $sort, $project etc
- Operators may include: $sum, $avg, $min, $max, $count etc
- Output of the previous stage becomes input to the next stage

Aggregation Example

- $match:
  - Similar to WHERE in SQL
  - db.orders.aggregate([{ $match: { status: “Pending" } }]);
- $group:
  - Similar to GROUP BY in SQL
  - db.orders.aggregate([{ $group: { _id: "$customerId", totalAmount: { $sum: "$amount" } } }]);
- $project
  - Similar to SELECT in SQL
  - db.orders.aggregate([{ $project: { _id: 0, customerId: 123, amount: 1 } }]);

Aggregation Example

- $sort:
  - Similar to ORDER BY in SQL
  - db.orders.aggregate([{ $sort: { amount: -1 } }]);
- $limit:
  - Similar to LIMIT in SQL
  - db.orders.aggregate([{ $limit: 5 }]);
- $unwind:
  - Useful when dealing with arrays
  - db.orders.aggregate([{ $unwind: "$items" }]);
- $lookup
  - Similar to JOIN in SQL
  - db.orders.aggregate([{ $lookup: { from: "products", localField: "productId", foreignField: "_id", as: "productDetails" } }]);

Aggregation Example

Aggregate Query

```
db.sales.aggregate([
{ $group: { _id: "$customerId", totalSales: {
$sum: "$amount" } } },
{ $sort: { totalSales: -1 } }
]);
```

Output

```
[
        { "_id": 1, "totalSales": 600 },
        { "_id": 2, "totalSales": 500 }
]
```

```
{   "_id":
ObjectId(“4f67bdf6c8a8b364f7b6e9f7"),
"customerId": 1,
"amount": 600,
 "date": ISODate("2020-01-01")},


{   "_id":
ObjectId("4f67bdf6c8a8b364f7b6e9f7 "),
"customerId": 2,
 "amount": 500,
 "date": ISODate("2020-02-01")
}
```

Some extra resources

- https://www.w3schools.com/xml/
- https://aws.amazon.com/compare/the-difference-between-json-xml/
- https://www.json.org/json-en.html
- https://www.mongodb.com/resources/basics/json-and-bson
- https://www.w3schools.com/mongodb/
- https://www.mongodb.com/docs/manual/