Università degli Studi di Messina

Data Science

Big Data Acquisition

-

Data Models

Prof. Daniele Ravi

dravi@unime.it

The CSV ecosystem

- CSV files are used in many fields due to its simplicity, portability, and widespread support across different platforms.
- CSV is a simple, human-readable data format commonly used for storing tabular data
- Each line in the file corresponds to a row in a table, and each field within that row is separated by a delimiter (commonly a comma).
- A text file containing rows of data, with values separated by the delimiter
- Typically, the first row contains column headers, which describe the type of data in each field.
- Different programming languages offer libraries and tools to read, write, and manipulate CSV files.

The CSV ecosystem

Cleaning and Preprocessing:

- Tools like `pandas` (Python) allow users to:
  - Handling missing values
  - Renaming columns
  - Filtering rows
  - Perform calculations
  - Merge different CSV files

CSV files are often used for:

- Data analysis
- Perform statistical analysis
- Generate visualizations

Applications of CSV Files

- Data Exchange: exporting and importing data between applications, databases, or systems (e.g., spreadsheets, databases, machine learning datasets).
- Data Sharing: They are lightweight, human-readable, and can be easily shared or emailed.
- Data Archiving: often used for long-term storage of tabular data.

The CSV ecosystem

Advantages:

- **Simplicity:** Very simple format where data is separated by commas. Easy to create, read, and parse.

- **Compact:** CSV files are typically very small in size, which makes them fast to load and transmit.

- **Widely supported:** Many applications (e.g., spreadsheets, databases) support importing/exporting CSV files.

- **Good for tabular data:** Excellent for representing flat, two-dimensional data such as rows and columns.

The CSV ecosystem

Disadvantages:

- **Lack of structure:** CSV files are limited to flat, tabular data. They cannot represent hierarchical or nested data structures.

- **No data types:** All data is stored as text, and interpreting numbers, booleans, or dates must be handled separately by the application.

- **Potential data corruption:** If data fields contain commas, line breaks, or special characters, they need to be escaped or quoted, which can complicate parsing.

- **No metadata:** CSV doesn't provide any way to store metadata like field names, types, or relationships between rows.

- **Large Data Files:** Handling large CSV files can be inefficient compared to binary formats or specialized databases.

The XML ecosystem

1. **XML (eXtensible Markup Language)**

- A flexible, text-based language used to store and transport data.
- It is human-readable and machine-readable, designed to represent data with a clear hierarchy of elements and attributes.
- DTD (Document Type Definition) and XSD (XML Schema Definition) are both used to define the structure and rules for XML documents
- They have several key differences in terms of features, expressiveness, and capabilities.

DTD (Document Type Definition)

- Uses a simpler, older syntax based on SGML (Standard Generalized Markup Language). It has its own specific, non-XML syntax.

XSD (XML Schema Definition)

- Written in XML syntax, which makes it easier to work with using XML tools, and more consistent with the XML document itself.
- It offers stronger validation than DTD, including data type constraints and inheritance.
- Used to ensure that the XML document adheres to the defined schema structure.

The XML ecosystem

Feature	DTD	XSD
Data Types	Limited (all text-based)	Rich data types (string, int, date, etc.)
Namespaces	Not supported	Fully supported
Syntax	Non-XML syntax	XML-based syntax
Customization	Limited	Highly customizable
Validation Power	Basic structure validation	Powerful, supports complex validations
Modularity and Reuse	Limited	Extensive
Ease of Use	Simple but limited	More complex, but much more powerful

The XML ecosystem

2. Namespace

- A way to avoid naming conflicts by qualifying element and attribute names in XML with a unique namespace identifier (usually a URI).
- Namespaces are useful when combining XML documents from different sources or defining multiple schemas in a single document.
- Declared using the `xmlns` attribute, e.g., `<book xmlns="http://example.com/books">`.

3. Element

- Elements are the primary building blocks of XML documents, representing structured data.
- An element consists of an opening tag, content, and a closing tag, e.g., `<title></title>`
- Elements can contain other elements (nested elements), attributes, or text.

The XML ecosystem

4. Attribute

- Attributes provide additional information about an element in the form of name-value pairs.
- They appear within the opening tag of an element, e.g., `<book id="123" title="XML Guide">`.
- Unlike elements, attributes cannot contain other elements or mixed content.

5. Character (Text)

- The actual textual content within an element
- XML documents primarily consist of characters (text) structured within elements.
- Example: `<message>Hello, World!</message>`.
- Text can be mixed with elements and attributes, known as "mixed content."

6. Comment

- Comments in XML are enclosed in `<!-- -->` and are ignored by XML parsers.
- They are used to add human-readable notes within the XML document.

The XML ecosystem

7. Querying

- XPath and XQuery allow querying and filtering of XML documents for specific data.
- You can select, retrieve, and manipulate nodes, attributes, and text using these languages

- **XPath (XML Path Language)**
- A query language used to navigate through elements and attributes in an XML document.
- It is used to locate specific parts of the XML document using path expressions.

- **Xquery**
- A query language used to retrieve and manipulate data from XML documents, similar to how SQL queries relational databases.

The XML ecosystem

8. Validation

- Ensures that an XML document conforms to its DTD or XML Schema (XSD) rules.
- Validation is crucial for ensuring that the data structure adheres to predefined rules

9. Unparsed Entity Notation Document

- Refers to the use of external resources (e.g., images or other media) within an XML document that cannot be parsed by the XML parser.
- These resources are declared in the DTD using the `NOTATION` mechanism, which specifies the format and type of the entity, but the entity itself remains outside the scope of XML parsing.

The XML ecosystem

Advantages:

- **Hierarchical structure:** Can represent complex, nested data with parent-child relationships.
- **Customizable tags:** Users can define their own tags, making it highly flexible for a variety of data types.
- **Self-descriptive:** Each piece of data is wrapped in descriptive tags, making it human-readable and self-explanatory.
- **Supports attributes:** Data can be stored in both elements and attributes, giving flexibility in representation.
- **Wide adoption:** Used in many industries for data interchange, especially where document-like structures are needed (e.g., SOAP for web services).

The XML ecosystem

Disadvantages:

- **Verbose:** XML files can be large due to the extensive use of opening and closing tags, which increases storage and bandwidth requirements.

- **Complexity:** Parsing XML can be more complicated and slower compared to JSON or CSV.

- **No native data types:** Everything is stored as a string, so numeric and boolean types must be interpreted by the application.

- **Overhead for small datasets:** Due to its verbose nature, XML is less efficient for representing small or simple datasets.

The JSON ecosystem

- A text-based format for representing structured data.
- Data is organized in key-value pairs using a structure of arrays and objects.
- Easy for humans to read and write
- Easy for machines to parse and generate.
- It's widely used in web development and APIs for data transmission due to its simplicity and compatibility with most programming languages.
  - JSON is the most common data format for RESTful APIs, where it is used for transmitting data between clients and servers.
  - API endpoints often send responses in JSON format, and clients (e.g., web browsers, mobile apps) parse these responses to display or process the data.

The JSON ecosystem

1. JSON Schema

- Is a vocabulary that allows you to validate the structure of JSON data.
- It defines the rules for the structure, properties, required fields, data types, and constraints of JSON documents.
- It serves a similar purpose to XML's DTD or XSD but for JSON. You can use it to ensure that JSON data follows a defined structure.

2. JSON Validation

- Checking if a JSON document complies with the rules defined in a JSON Schema.
- JSON data can be validated at different stages (e.g., before being stored, after retrieval, or during data transmission).

The JSON ecosystem

3. Parsing JSON

- JSON parsing involves converting JSON data (typically in string format) into a data structure that a programming language can process, such as an object or array.
- Most programming languages have built-in functions or libraries for parsing JSON data.
- I.e. in Python, `json.loads()` performs the parsing

4. Serializing JSON

- Serialization refers to converting a data structure (like an object or array) into a JSON-formatted string.
- This is often necessary when sending data from a server to a client or saving data to a file in a structured format.
- I.e. In Python, you use `json.dumps()`

The JSON ecosystem

7. JSON Querying

- **Using languages or tools that support JSON natively:**
  - JavaScript, where you can directly interact with JSON as objects or arrays.
  - NoSQL databases like MongoDB use JSON-like documents to store and query data.
  - SQL databases with JSON support (like PostgreSQL and MySQL) allow querying JSON data using specialized functions like `JSON_EXTRACT()`.

- **JSONPath**
  - Is an expression language used to navigate and query JSON data.
  - It allows for the selection of specific parts of JSON documents, much like XPath does for XML.
  - JSONPath enables filtering, array slicing, and retrieving nested data in JSON objects.

JSON vs. XML

- **Simplicity**: JSON is simpler and more compact than XML, using fewer characters to represent the same data structure.
- **Readability**: JSON is often easier for humans to read due to its familiar syntax, similar to JavaScript objects.
- **Data Types**: JSON supports primitive data types such as numbers, strings, booleans, arrays, and objects, while XML represents everything as a string.
- **Use Cases**: JSON is widely used in web development, APIs, and NoSQL databases, while XML is still preferred in cases where document structure validation (DTD/XSD) or more complex document structures are required.

The JSON ecosystem

Advantages:

- **Lightweight:** Less verbose than XML, reducing file size and improving transmission efficiency.
- **Native support for data types:** Directly supports numbers, booleans, arrays, and objects, making it easier for programming languages (especially JavaScript) to parse and use.
- **Human-readable:** Easy to read and write, even for beginners.
- **Interoperability:** Supported by many programming languages, making it a common format for web APIs and data interchange.
- **Nested structures:** Like XML, JSON supports hierarchical/nested data.

The JSON ecosystem

Disadvantages:

- **Limited metadata support:** Unlike XML, JSON doesn’t allow the use of attributes, which limits its ability to describe data relationships or structure beyond basic objects and arrays.

- **No comments:** JSON doesn't support comments, which can make maintaining large files harder.

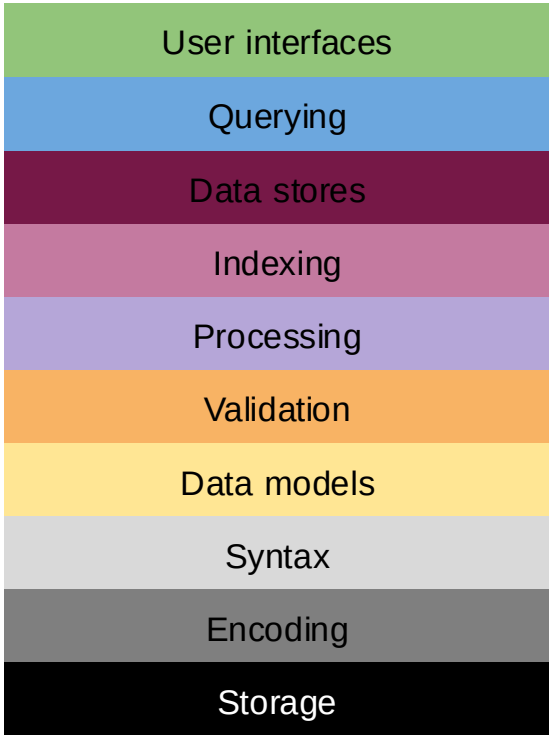
- **Less flexible for document-centric data:** While JSON is great for data interchange, it’s not as well-suited for representing rich document structures as XML.

Summary

- Each format has its strengths and weaknesses, and the best choice depends on the specific use case—whether you need simple tabular data, structured and nested data, or document-like data with detailed metadata.

Format	Advantages	Disadvantages
XML	Hierarchical structure, customizable, self-descriptive, widely adopted	Verbose, complex, no native data types, less efficient for small datasets
JSON	Lightweight, native data types, human-readable, nested structures, widely supported	Limited metadata support, no comments, less suited for document-centric data
CSV	Simple, compact, widely supported, good for tabular data	Lack of structure, no data types, potential data corruption, no metadata

Data Technology Stack



10 Principle when designing systems for big data

- When designing systems for big data, it is essential to follow some principles:

1. Learn from the past
2. Keep the Design Simple
3. Modularize the Architecture
4. Homogeneity in the large
5. Heterogeneity in the small
6. Abstract logical model from its physical implementation
7. Shard the data
8. Replicate the data
9. Buy lots of cheap hardware
10. Parallelization and Batch Processing

1. Learn from the past

- Not to underestimate the valuable principles and practices established by **past technologies**
- Many core concepts from traditional databases, such as **data independence**, **relational algebra**, and operations like **selection** and **projection**, remain highly relevant.
- While we may need to redesign systems to fit the distributed nature of big data (e.g., clusters), these foundational principles are still critical for ensuring robust and efficient data management.
- One common mistake is the tendency to disregard well-established ecosystems—such as the **XML-based ecosystem**—in favor of newer, simpler solutions like JSON.

1. Learn from the past

- While JSON provides simplicity by not requiring a schema or query language, it’s important to recognize that, in some contexts, the capabilities provided by XML—such as **schema validation**, **structured querying**, and **hierarchical navigation**—are actually necessary.

- A more balanced approach would be to adopt simpler ecosystems like JSON while incorporating some key features we've learned from XML.

- For instance, while JSON doesn't inherently require schemas or query languages, it can support both through tools like **JSON Schema** for validation and **JSONPath** for querying.

- When designing new systems, it's essential to **study existing literature** and **leverage lessons from past technologies**.

  This approach not only saves time but also helps avoid repeating mistakes that have already been addressed in earlier systems.

2. Keep the Design Simple

- **Scalability** is one of the core challenges in big data, and one of the most effective ways to scale systems is by keeping the design as simple as possible.

- Complex systems are harder to manage, optimize, and scale, particularly when working with massive amounts of data across distributed architectures.

- A good example of this is **storing big data** in structures like flat tables, where we have **keys** and their associated **values**.

- This minimalistic approach is foundational to many NoSQL databases and distributed systems. Flat data models allow for fast lookups, easier partitioning, and distribution across clusters.

- However, while this simplicity is advantageous, it also introduces challenges when the number of keys reaches **billions or even trillions**—a common scenario in big data environments.

2. Keep the Design Simple

- This is where formats like **JSON** have gained popularity.
- JSON is conceptually closer to relational tables in many respects because it provides a **key-value structure** but also supports **hierarchical, nested data**.
- JSON's simplicity makes it ideal for storing and transmitting data in distributed systems, which need to handle varying and evolving data structures.
- For instance, keeping data models flat and avoiding excessive nesting or deeply hierarchical structures helps ensure that systems can scale horizontally, making them more efficient and easier to maintain.
- Furthermore, **simplified data models** reduce the cognitive load for developers, making the system easier to understand, debug, and optimize.

3. Modularize the Architecture

- Modular design breaks down the system into **distinct layers** or components that can be developed, maintained, and scaled independently.
- Each layer of the architecture should focus on a **specific responsibility**, abstracting away complexity from other layers, and exposing only necessary functionality to the layers above.
- This makes the system easier to understand, manage, and evolve.
- Focus on **one layer at a time**, assuming the underlying layers are already built and functioning correctly.
- This **separation of concerns** improves flexibility:
  - Changes in one layer (e.g., improvements to storage or data access) don't necessarily affect other layers.
  - Allows you to **swap out technologies** or update individual components without disrupting the entire system.

3. Modularize the Architecture Example: HBase and HDFS

- **HBase** is a distributed, column-oriented database built on top of **HDFS** (Hadoop Distributed File System).

- **HDFS Layer**
  - **P**rovides scalable storage by distributing data across **thousands of machines**.
  - It is optimized for storing massive amounts of data **in a fault-tolerant way**.
  - Is designed for **batch processing** rather than **random access**—you can append data to files or write files, but you cannot efficiently read or modify small parts of the data.
  - This is because HDFS writes files in large blocks and does not support random reads at the byte level efficiently.

- **HBase Layer**
  - **HBase** was introduced to solve the **HDFS** limitations.
  - HBase provides a **random-access layer** on top of the batch-oriented HDFS.
  - It also organizes data into a **tabular structure** with rows and columns, which adds a level of structure, allowing for more efficient data retrieval.

3. Modularize the Architecture Example: HBase and Hive

- **Hbase:**
- **Random access:** It provides the ability to quickly access individual rows of data, which is critical for many real-time applications
- **Table-like structure:** HBase stores data in a schema-less, column-family-based model that is similar to a table, allowing indexing and efficient data retrieval
- HBase exposes APIs for basic operations (GET, PUT, DELETE, SCAN), which allow for flexible interaction with the data, though it does not natively offer.

- **Hive:**
- **SQL-like query capabilities:** It provides a higher-level abstraction over HBase and HDFS by allowing users to write queries in **SQL-like syntax**.
- Hive translates the SQL queries into **MapReduce** jobs or other processing frameworks, which are executed across the cluster, eventually reading and writing data from HBase and HDFS.

3. Modularize the Architecture Example: HBase and Hive

- One might wonder why HBase doesn’t expose higher-level interfaces like SQL natively and instead provides an API with basic operations such as GET, DELETE, and PUT.

- This is because HBase's purpose is to offer **random access** to large datasets stored across a distributed system, which HDFS alone cannot do efficiently.

3. Modularize the Architecture

- **HDFS:**
  Provides scalable, distributed storage for massive datasets, but optimized for sequential access (batch processing).

- **HBase:**
  Adds random access and tabular structures on top of HDFS, enabling efficient data retrieval and updates.

- **Hive (or similar tools):**
  Provides a SQL-like interface, translating high-level queries into lower-level operations that flow down the stack to HBase and HDFS.

Benefits of Modular Architecture

- **Isolation of Concerns:**
  Each architecture layer (HDFS, HBase, Hive) has specific responsibilities, simplifying maintenance and evolution. HBase developers don't need to manage HDFS, and Hive users don't need to understand HBase.

- **Interchangeable Components:**
  If a better storage system replaces HDFS, minimal changes are needed for HBase or Hive.

- **Scalability and Flexibility:**
  Each layer scales independently. HDFS handles large data storage, HBase enables random access, and Hive offers high-level querying.

- **Abstraction for Simplicity:**
  Layers hide complexity. Hive users interact with SQL-like queries, avoiding HBase's low-level API and HDFS intricacies.

4. Homogeneity in the large & 5. Heterogeneity in the small

- **Homogeneity at Large Scale:**
- Uniform data makes handling trillions of rows and massive datasets easier, allowing efficient distribution and processing across clusters.

- **Heterogeneity at Smaller Scale:**
- Real-world data is diverse, leading to different formats, missing values, and unstructured or semi-structured data, making small-scale data more complex.

- **Balancing with Denormalization:**
- Denormalization helps manage small-scale variability (e.g., missing values, different data types) while maintaining a structure optimized for scalability.
- This approach provides the flexibility to handle diverse data at the micro-level while ensuring smooth scaling at the macro-level.

6. Abstract logical model from its physical implementation

- **Separation of Physical and Logical Layers:**
  Data independence allows for managing complex systems efficiently by decoupling the physical storage from logical operations.

- **Logical Layer Interaction:**
  Users interact with APIs at the logical layer, abstracted from the complexities of the underlying physical layer (e.g., storage, distribution).

- **Designing Effective APIs:**
  The challenge lies in creating APIs that cleanly isolate users from physical layer changes, ensuring that optimizations or changes (e.g., storage formats) don't impact data access.

- **Flexibility and Scalability:**
  This separation allows systems to scale and evolve independently across layers, maintaining performance while adapting to growing data demands.

7 .Shard the data

- In big data systems, homogeneity at a large scale enables efficient data partitioning, often referred to as **sharding**.

- **Sharding for Partitioning:**
- Homogeneity at scale allows for sharding—dividing data into smaller, manageable pieces (chunks, blocks, partitions) that can be evenly distributed across a cluster.
- This supports parallel processing and enhances scalability.

- **Batch Processing Efficiency:**
- Sharding is especially useful for batch processing, where large volumes of data are processed in bulk.
- Each node in the cluster processes a portion of the data independently, leading to faster and more efficient overall performance.

- **Sequential vs. Random Access:**
- Big data systems avoid random access patterns, as they create bottlenecks at scale. Instead, data is processed sequentially in a uniform manner, optimizing throughput and maximizing cluster resource utilization.

8. Replicate the data

- In distributed storage systems like Hadoop's HDFS (Hadoop Distributed File System), replication is a core principle.

- **Data Replication for Fault Tolerance:**
  Data is split into partitions and replicated across multiple nodes (typically 3 copies), ensuring fault tolerance in case of hardware failure.

- **Avoid Over-Replication:**
  Excessive replication wastes storage and adds complexity, so it's important to balance replication levels.

- **HDFS Manages Replication:**
  Systems like HBase, which run on top of HDFS, don’t need to replicate data again, as they benefit from HDFS's native replication.

9. Buy lots of cheap hardware

- **Horizontal Scaling (Scaling Out)**
  - Purchase **many inexpensive machines** instead of a single powerful one.
  - Also known as **"scaling out"**.
  - Distribute workload across multiple machines for **cost-effectiveness**.

- **Benefits of Horizontal Scaling**
  - Machines work as a **cluster** in data centers.
  - Increases **capacity** and **fault tolerance**.
  - If one machine fails, tasks are **redistributed** to others.
  - More scalable and **lower cost** than vertical scaling.

- Distributed systems like Hadoop, Spark, and NoSQL databases (e.g., Cassandra, HBase) are designed to take advantage of this approach.

10. Parallelization and Batch Processing

- **Parallelization**
  - **Objective**: Balance **capacity** (data volume) with **speed** (data processing rate).
  - **Method**: Distribute data processing across multiple nodes.
  - **Result**: Increased **throughput** by reading and processing data simultaneously.

- **Throughput & Latency**
- **Throughput**: The amount of data processed over a specific period.
  - Higher throughput = ability to handle large data volumes quickly.

- **Latency**: The time delay for a system to respond to a request.
  - Even with high throughput, **latency** may remain significant.
  - Affects **real-time** and **near-real-time** operations.

10. Parallelization and Batch Processing

- **Flushing**
- Process of writing data from **temporary memory (buffers)** to **permanent storage (disk)**.
- Instead of writing small pieces, flushing writes **larger data blocks** at once.
- **Benefit**:
  - **Reduces the number of write operations**.
  - Less overhead and better system resource usage.
  - Helps lower overall **system latency**.

- **Compacting**
- Combines small, scattered data files into **fewer, larger files**.
- Reorganizes underutilized data blocks to **maximize space efficiency**.
- **Benefit**:
  - Reduces data fragmentation, improving **read performance**.
  - Lowers latency by minimizing the inefficiencies of **random access**.

10. Parallelization and Batch Processing

- **HDFS (Hadoop Distributed File System)**
  - **Designed for sequential access**, not random access.
  - **Random access** increases latency as data blocks must be retrieved from different nodes, causing slower data retrieval.

- **Batch Processing Approach**
- **Data split into blocks** across different nodes.
  - Nodes **read and process** data in parallel, improving **throughput** (data handled).
- **Sequential access**: HDFS processes **larger blocks** efficiently, unlike random access which is slower.

- **Batch Processing**: Groups large datasets together, processing and writing them in **larger blocks**.
  - Reduces overhead from frequent, smaller operations.

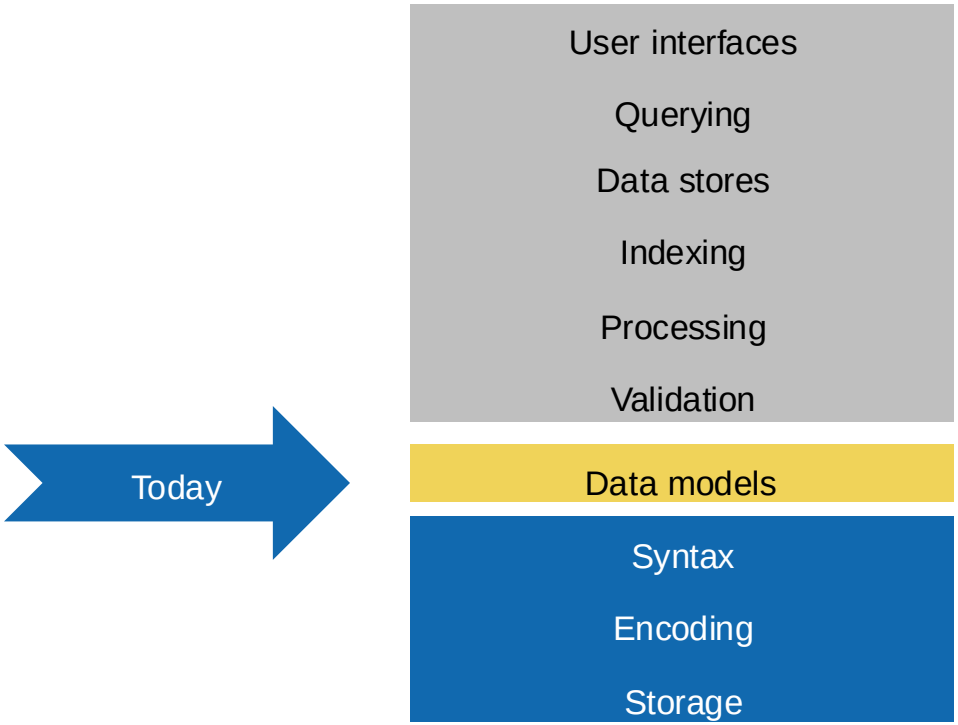
- **Latency Improvement**
- Even though individual batches may take time, overall system **latency is reduced** by avoiding frequent small operations.

10. Parallelization and Batch Processing

- **Parallelization**:
  - Solves the **capacity-throughput mismatch** by distributing data and processing across multiple machines.
  - **Outcome**: Faster data reading and processing as tasks are handled simultaneously across nodes.

- **Batch Processing**:
  - Addresses the **throughput-latency mismatch** by minimizing delays through efficient handling of large datasets.
  - **Outcome**: Fewer, larger operations instead of frequent, small ones, leading to better performance and lower latency.

Where we are


Today
User interfaces
Querying
Data stores
Indexing
Processing
Validation
Data models
Syntax
Encoding
Storage

Syntax vs. Data Models

- **Definition of Syntax**
  Syntax refers to the way data is stored in a textual format.
- **Forms of Data**
  Data can take various forms, including:
  - **Tables** (e.g., CSV files)
  - **Hierarchical Formats** (e.g., XML, JSON)

- **Logical Interpretation**
  While represented as plain text, these formats are interpreted in a more structured manner.

- **Example: CSV Files**
  - Data is stored as raw text.
  - Interpreted using a relational data model:
    - **Rows and Columns** represent structured tables.

Syntax vs. Data Models

- **Tree-Like Structure**
  - XML and JSON store data in a hierarchical format.
- **Recursive Traversal**
  - Requires recursive traversal to interpret:
    - **Parent-Child Relationships** within the data.
- **Structured Data Models**
  - Though represented as text, they should be viewed as structured data models, similar to CSV files.
- **Key Takeaway**
  - Regardless of textual syntax, always consider these formats as structured data representations during reading and processing.

Syntax vs. Data Models

Logical view

Data Model

ID	Last name	First name	Theory
1	Einstein	Albert	General, Special Relativity
2	Gödel	Kurt	"Incompleteness" Theorem

This is a data model

Physical view

Syntax

```
ID,Last name,First name,Theory,
1,Einstein,Albert,"General, Special Relativity"
2,Gödel,Kurt,""""Incompleteness"" Theorem"
```

JSON Values

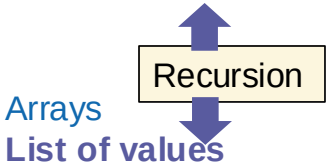
Strings

Numbers

Booleans

Null

Objects
String-to-Value map



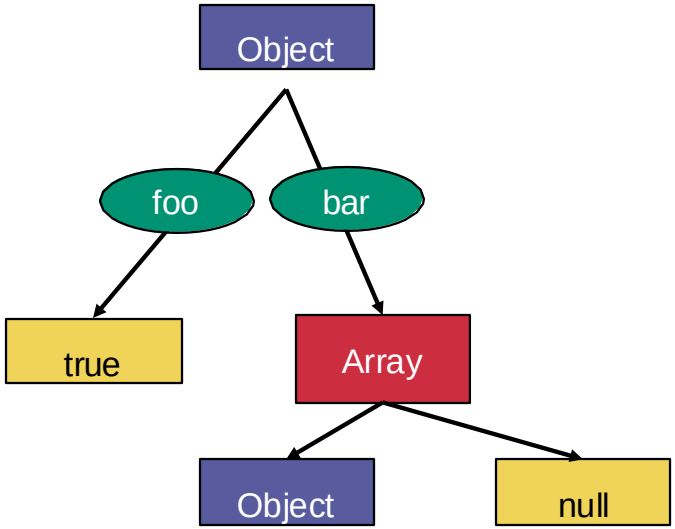
Arrays
List of values

Atomic values
Leaves of the tree

Structured values
Recursion

Tree-based visual model

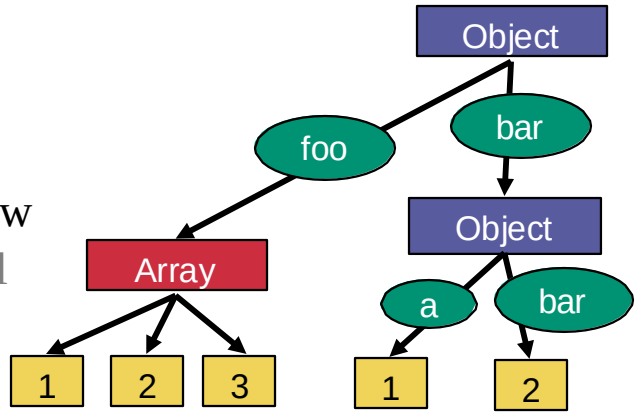
```
{
  "foo" : true,
  "bar" : [
    {
      "foobar" : "foo"
    },
    null
  ]
}
```



Syntax vs. Data Models

Logical view

Data Model



Physical view

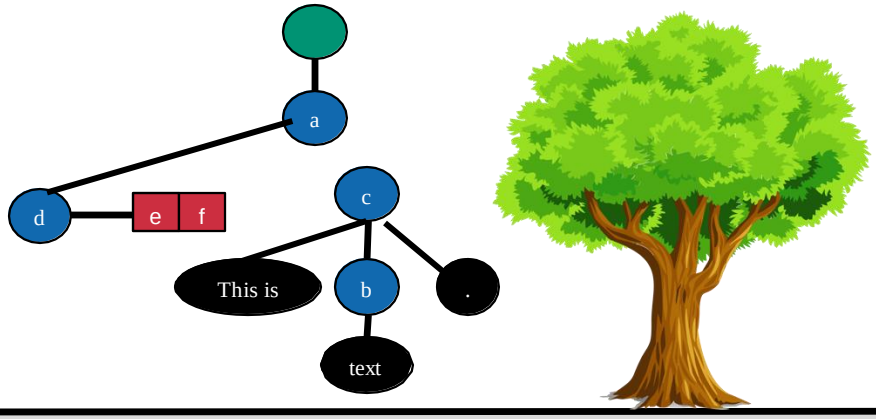
Syntax

```
{
  "foo" : [ 1, 2, 3 ],
  "bar" : { "a" : 1, "bar" : 2 }
}
```

Syntax vs. Data Models

Logical view

Data Model

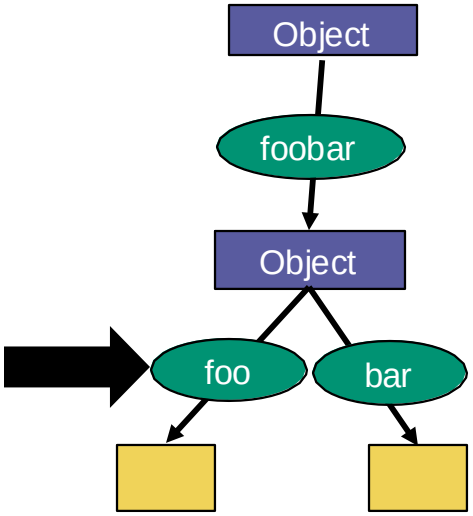


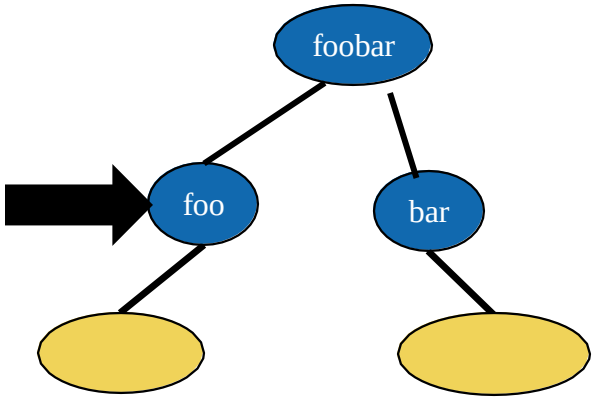
Physical view

Syntax

```
<a>
  <d e="f"/>
  <c>This is <b>text</b>.</c>
</a>
```

Edge vs. Node labeling





JSON

Labels are on the **edges**

XML

Labels are on the **nodes**

XML Information Set

Information Set

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE metadata>
<metadata>
    <title
        language="en"
        year="2019"
        >Systems Group</title>
    <publisher>ETH Zurich</publisher>
</metadata>
```

The 11 XML Information Items

- Document
- Element
- Attribute
- Processing Instruction
- Character
- Comment
- Namespace
- Unexpanded Entity Reference
- DTD
- Unparsed Entity
- Notation

The 4 XML (most important)

Document
Element
Attribute
Processing Instruction
Character (Text)
Comment

Namespace
Unexpanded Entity Reference
DTD
Unparsed Entity
Notation

Document Information Items

Document Information Item doc
[children] Element Information Item metadata
[version] 1.0

The Document Information Item is always present (even if the optional DOCTYPE is missing)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE metadata>
 <metadata>
     <title
         language="en"
         year="2019"
         >Systems Group</title>
    <publisher>ETH Zurich</publisher>
</metadata>
```

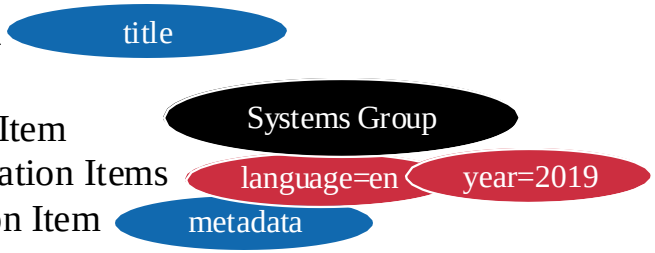
Element Information Items

Element Information Item metadata
[local name] metadata
[children] Element Information Items title publisher
[attributes] <empty>
[parent] Document Information Item doc

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE metadata>
 <metadata>
     <title
         language="en"
         year="2019"
         >Systems Group</title>
    <publisher>ETH Zurich</publisher>
</metadata>
```

Element Information Items

Element Information Item
[local name] title
[children] Text Information Item
[attributes] Attribute Information Items
[parent] Element Information Item




```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE metadata>
 <metadata>
     <title
         language="en"
         year="2019"
         >Systems Group</title>
    <publisher>ETH Zurich</publisher>
</metadata>
```

Element Information Items

Element Information Item publisher
[local name] publisher
[children] Text Information Items ETH Zurich
[attributes] Attribute Information Items <empty>
[parent] Element Information Item metadata

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE metadata>
 <metadata>
     <title
         language="en"
         year="2019"
         >Systems Group</title>
    <publisher>ETH Zurich</publisher>
</metadata>
```

Attribute Information Items

Attribute Information Item year=2019
[local name] year
[normalized value] 2019
[owner element] Element Information Item title

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE metadata>
 <metadata>
     <title
         language="en"
         year="2019"
         >Systems Group</title>
    <publisher>ETH Zurich</publisher>
</metadata>
```

Attribute Information Items

Attribute Information Item language=en
[local name] language
[normalized value] en
[owner element] Element Information Item title

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE metadata>
 <metadata>
     <title
         language="en"
         year="2019"
         >Systems Group</title>
    <publisher>ETH Zurich</publisher>
</metadata>
```

Text Information Items

Text Information Item Systems Group

[characters] S y s t e m s <space> G r o u p

[owner element] Element Information Item title

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE metadata>
 <metadata>
     <title
         language="en"
         year="2019"
         >Systems Group</title>
    <publisher>ETH Zurich</publisher>
</metadata>
```

Text Information Items

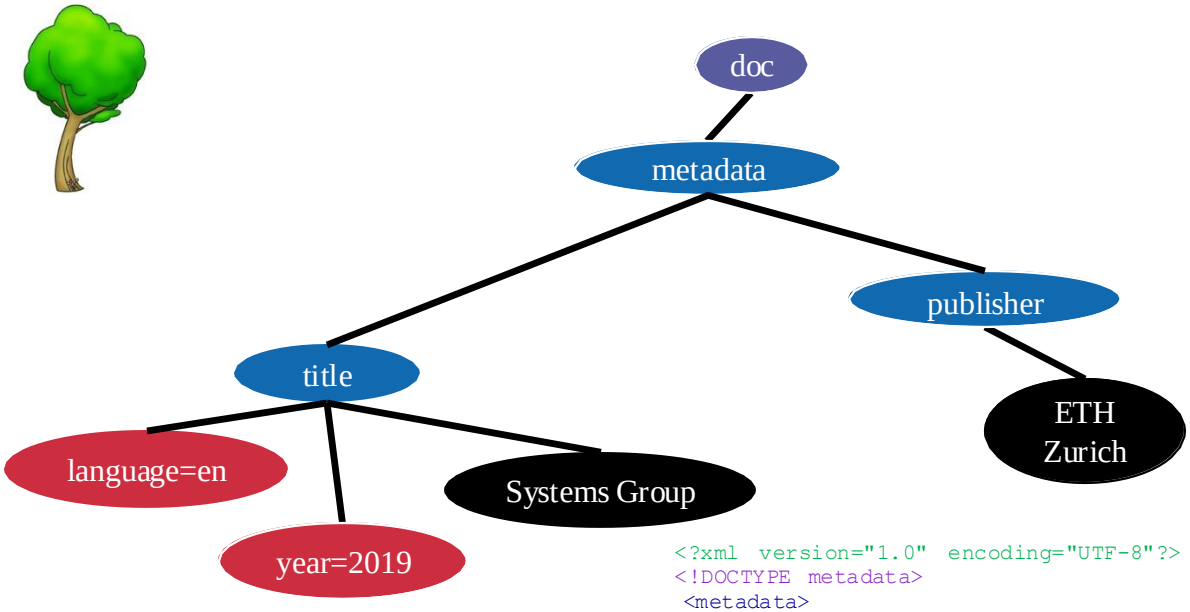
Text Information Item ETH Zurich

[characters] E T H <space> Z u r i c h

[owner element] Element Information Item publisher

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE metadata>
 <metadata>
     <title
         language="en"
         year="2019"
         >Systems Group</title>
    <publisher>ETH Zurich</publisher>
</metadata>
```

XML Infosetthe tree




```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE metadata>
 <metadata>
     <title
         language="en"
         year="2019"
         >Systems Group</title>
    <publisher>ETH Zurich</publisher>
</metadata>
```

Validation

Burak Cakmak / 123RF Stock Photo

Validation

- **Well-formed**: Adheres to the basic syntax rules of JSON or XML.

- **Validation**: (applies to XML or JSON with schema)
- Checks if the structure and content conform to a defined schema or set of rules.
- JSON requires a separate schema (like JSON Schema), while XML uses DTD or XSD.

Well-Formed XML VS Valid XML:

- **Well-Formed XML**
  - An XML document is **well-formed** if it follows basic syntax rules:
  - Proper opening and closing of tags.
  - Tags must be **properly nested**.
  - There must be **one root element** that encapsulates the entire document.
  - Attribute values must be **quoted**.
  - No unescaped special characters (e.g., &, <).

- **Valid XML**
  - An XML document is **valid** if it conforms to a specific **Document Type Definition (DTD)** or **XML Schema Definition (XSD)**.
  - **Valid XML** is always well-formed, but not all well-formed XML is valid.
  - Validation checks if the XML adheres to specific rules on structure, element names, data types, and attribute values.

- **Example of Validation**
- A DTD or XSD might require that the <age> element be a **numeric value**.
- Validation ensures the XML conforms to the structure and constraints defined in the schema.

Well-Formed vs Valid JSON

- **Well-Formed JSON**:
  - Adheres to basic syntax rules:
    - **Key-value pairs** within curly braces {} for objects.
    - **Square brackets** [] for arrays.
    - Proper use of **commas** and **colons** to separate elements and key-value pairs.
    - **Double quotes** for keys and string values.
    - **Valid data types**: strings, numbers, booleans, arrays, objects, null.
- **Validation in JSON**:
  - JSON has no built-in validation, but can be validated using **JSON Schema**.
  - **JSON Schema** defines structure, data types, and constraints.
  - **Example**:json

```
{
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "age": { "type": "integer" }
  },
  "required": ["name", "age"]
}
```

  - Ensures **"name"** is a string and **"age"** is an integer.

Well-Formed vs Valid XML

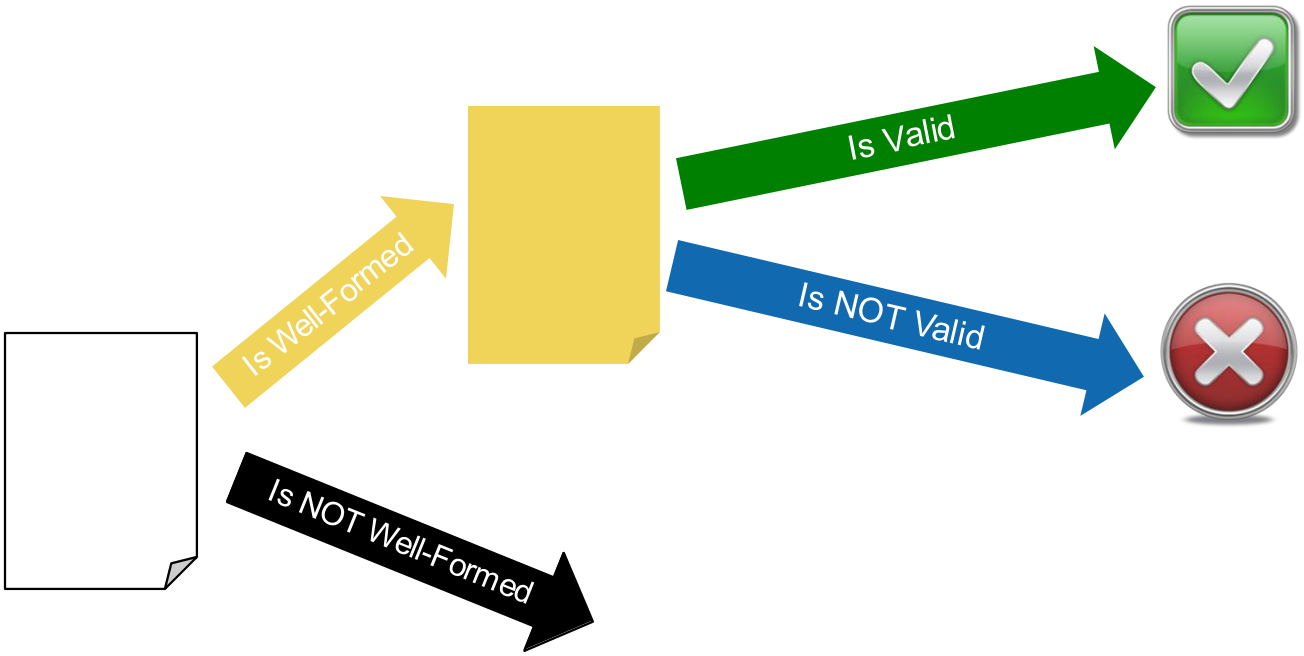
- **Well-Formed XML**:
  - Follows basic syntax rules:
    - Properly nested and closed tags.
    - One root element, quoted attributes, no unescaped special characters.

- **Valid XML**:
  - Requires **DTD** or **XSD** to validate structure, element names, and data types.
  - A **valid XML** is also **well-formed**.

- **Summary**
  - **Well-formed**: Adheres to basic syntax rules (JSON or XML).
  - **Validation**: Ensures structure and content conform to a defined schema (JSON Schema for JSON; DTD/XSD for XML).

Validation: The Pipeline

Validation



Heterogeneous vs. homogeneous datasets

No schema
(heterogeneous)

With schema
(homogeneous)

Without schema…

```
{
  "a" : 1,
  "b" : [ "foo", true, null, { "foo" : "bar" } ],
  "c" : {
    "d" : { "foo" : null },
    "e" : [ 1, 2, [ 3, 4 ] ],
    "f" : 3.14
  }
}
```

With schema…

```
{
  "a" : 1,
  "b" : true,
  "c" : [
    { "foo" : "bar1",  "bar" : [ 1, 2 ] },
    { "foo" : "bar2",  "bar" : [ 3, 4, 5 ] },
    { "foo" : "bar3" }
  ]
}
```

Introduction to Schemas: What is a Schema?

- A schema is a formal definition that describes the structure of a document.
- It specifies the types of data and constraints for the data elements.
- Why Use Schemas?
  - Ensures data consistency and integrity.
  - Facilitates validation of data against defined rules.

Creating a JSON Schema

- **Key Components:**
  - `$schema`: Specifies the version of the JSON Schema.
  - `type`: Defines the data type (object, array, string, etc.).
  - `properties`: Lists the properties of an object.
  - `required`: Indicates which properties are mandatory.

- JSON Document:

```
{
  "name": "John Doe",
  "age": 30,
  "email": "john.doe@example.com"
}
```

- JSON Schema:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "age": { "type": "integer", "minimum": 0 },
    "email": { "type": "string", "format": "email" }
  },
  "required": ["name", "age", "email"]
}
```

Creating an XML Schema (XSD)

- **Key Components**:
  - `<xs:schema>`: Root element defining the schema.
  - `<xs:element>`: Defines individual elements.
  - `<xs:complexType>`: Describes complex data types.
  - `<xs:attribute>`: Specifies attributes for elements.

XML Document:

```
<person>
   <name>John Doe</name>
   <age>30</age>
   <email>john.doe@example.com</email>
 </person>
```

XML Schema (XSD):

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="person">
    <xs:complexType>
     <xs:sequence>
       <xs:element name="name" type="xs:string"/>
       <xs:element name="age" type="xs:integer"/>
       <xs:element name="email" type="xs:string"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
 </xs:schema>
```

Validation of JSON and XML Documents

- **JSON Validation:**
  - Use JSON Schema validators to check conformity.

- **XML Validation:**
  - Use XML Schema validators or tools like XMLSpy or Oxygen XML to validate XML against XSD.

- **Keep Schemas Updated:**
  - Regularly review and update schemas as data requirements evolve.
- **Document Schemas:**
  - Provide clear documentation for each schema to aid understanding.
- **Use Consistent Naming Conventions:**
  - Ensure names are descriptive and follow a consistent pattern.

- Schemas are essential for data validation and integrity in both JSON and XML.
- Understanding how to create and utilize schemas can significantly improve data management practices.

What is Oxygen XML?

- Oxygen XML is a powerful XML editor and authoring tool.
- Designed for XML development, schema validation, and content authoring.
- Supports various XML technologies and standards.

- **Key Features of Oxygen XML**
- XML Editing:
  - Intuitive user interface for easy navigation and editing.
  - Syntax highlighting and code folding.
  - Smart completion and suggestions for XML elements.

- Schema Support:
  - Validation against XML Schema (XSD), DTD, and Schematron.
  - Support for Relax NG and other schema languages.

What is Oxygen XML?

- **Collaboration Tools:**
  - Built-in version control for managing changes.
  - Integration with Git, Subversion, and other VCS.

- **Integration:**
  - Compatible with various IDEs (Eclipse, IntelliJ IDEA).
  - Connects to external tools and frameworks (e.g., XSLT, XQuery).

- **Extensibility:**
  - Customizable through plugins and extensions.
  - Supports custom transformation scenarios.