Università degli Studi di Messina

Data Science

Big Data Acquisition Syntax

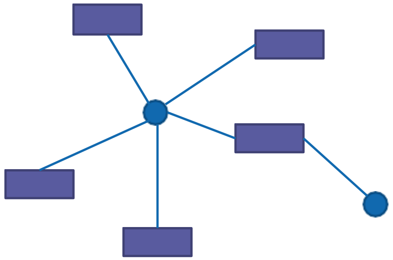
Prof. Daniele Ravi

dravi@unime.it

Data Shapes

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam vel erat nec dui aliquet vulputate sed quis nulla. Donec eget ultricies magna, eu dignissim elit. Nullam sed urna nec nisl rhoncus ullamcorper placerat et enim. Integer varius ornare libero quis consequat. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean eu efficitur orci. Aenean ac posuere tellus. Ut id commodo turpis.

Praesent nec libero metus. Praesent at turpis placerat, congue ipsum eget, scelerisque justo. Ut volutpat, massa ac lacinia cursus, nisl dui volutpat arcu, quis interdum sapien turpis in tellus. Suspendisse potenti. Vestibulum pharetra justo massa, ac venenatis mi condimentum nec. Proin viverra tortor non orci suscipit rutrum. Phasellus sit amet euismod diam. Nullam convallis nunc sit amet diam suscipit dapibus. Integer porta hendrerit nunc. Quisque pharetra congue porta. Suspendisse vestibulum sed mi in euismod. Etiam a purus suscipit, accumsan nibh vel, posuere ipsum. Nulla nec tempor nibh, id venenatis lectus. Duis lobortis id urna eget tincidunt.



Syntax of data

- **Syntax of data** refers to the structure, format, or organization of data in a module or dataset.
- Big data can come in various forms (structured, semi-structured, or unstructured), and each form has its own syntax.

1. **Structured Data Syntax:**

- Structured data follows a strict schema and is often organized in a tabular format like databases or spreadsheets.

- Examples include relational databases (SQL), where data is stored in tables with rows and columns. The syntax focuses on the use of fields, data types, and relationships.

Structured data

ID	Name	Age	Degree
1	John	18	B.Sc.
2	David	31	Ph.D.
3	Robert	51	Ph.D.
4	Rick	26	M.Sc.
5	Michael	19	B.Sc.

Syntax of data

2. Semi-Structured Data Syntax:

- Doesn’t follow a rigid schema but still contains tags or markers to separate elements.
- Common formats include JSON (JavaScript Object Notation) and XML (Extensible Markup Language).

3. Unstructured Data Syntax:

- Doesn’t have a predefined data model or organization.
- Examples include text files, images, audio, and video data. In big data systems, unstructured data is often processed using tools like Hadoop or NoSQL databases.

4. Key/Value Store Syntax:

- Key/Value stores (such as NoSQL databases) store data as key-value pairs. The syntax usually involves defining keys that correspond to particular values without needing a strict schema.

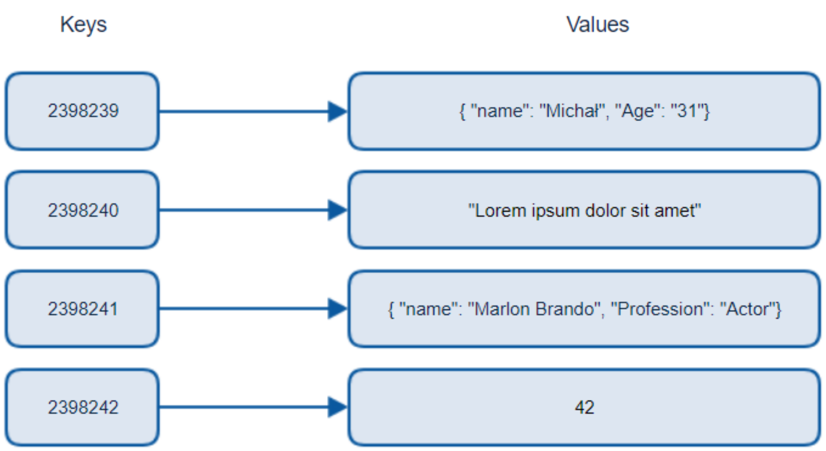
Unstructured data

The university has 5600 students. John’s ID is number 1, he is 18 years old and already holds a B.Sc. degree. David’s ID is number 2, he is 31 years old and holds a Ph.D. degree. Robert’s ID is number 3, he is 51 years old and also holds the same degree as David, a Ph.D. degree.

Semi-structured data

```
<University>
 <Student ID="1">
  <Name>John</Name>
  <Age>18</Age>
  <Degree>B.Sc.</Degree>
 </Student>
 <Student ID="2">
  <Name>David</Name>
  <Age>31</Age>
  <Degree>Ph.D. </Degree>
 </Student>
....
</University>
```

Key/Value Store Syntax



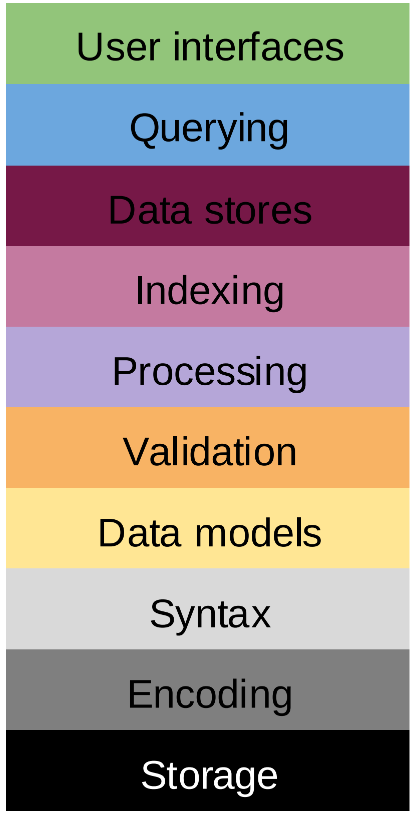
The stack: User Interfaces

This stack represents various layers of a **typical big data architecture**.

Each layer plays a crucial role in the lifecycle of big data, from user interaction and querying to storage and encoding.

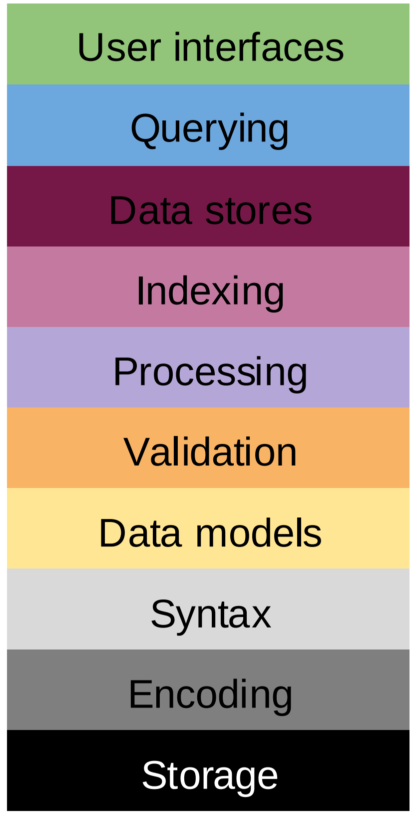
1. User Interfaces

- The user interface (UI) is how end users interact with the big data system.
- It provides the means for users to view, query, analyze, and visualize data, as well as manage the system.
- These interfaces can be graphical (GUI) or command-line (CLI).
- For big data, UIs could be dashboards, web-based interfaces, or applications.



The stack: Querying

- Querying refers to the process of retrieving specific data from a dataset by using a query language.
- For structured data, this is often done using SQL (Structured Query Language).
- For semi-structured and unstructured data, there are other query methods like NoSQL queries or custom data retrieval methods.
- Querying helps users extract meaningful insights or answer specific questions by filtering, aggregating, and manipulating the data.



Examples:

- SQL for relational databases
- SPARQL for RDF (Resource Description Framework) data
- NoSQL queries for databases like MongoDB or Cassandra

The stack: Data stores

- Data stores are the repositories where data is kept for future access, retrieval, and analysis.
- They are designed to store massive amounts of data efficiently and can be structured (relational databases), semi-structured (NoSQL databases), or unstructured (HDFS, cloud storage).

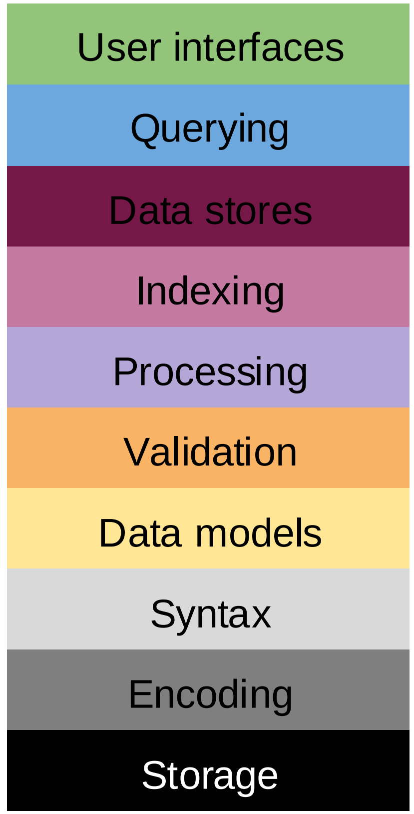
Types of Data Stores:

- Relational Databases: MySQL, PostgreSQL

- NoSQL Databases: MongoDB, Cassandra, Redis

- Distributed File Systems: Hadoop Distributed File System (HDFS)

- Cloud Storage: AWS S3, Google Cloud Storage

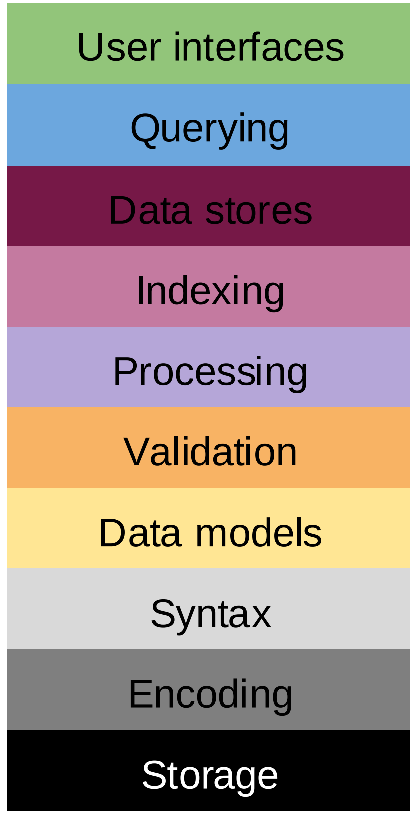


The stack: Indexing

- Indexing improves the speed and efficiency of data retrieval.
- An index is a data structure that allows for faster searches by organizing data in a way that allows for rapid querying.
- In the context of big data, indexing is crucial for handling large volumes of data and ensuring that queries are processed quickly.

Examples:

- Lucene/Solr/Elasticsearch for full-text indexing and search
- B-tree and hash indexes in relational databases
- Secondary indexes in NoSQL databases like Cassandra



The stack: Processing

- Processing refers to transforming, analyzing, and manipulating large datasets to derive insights or prepare the data for analysis.

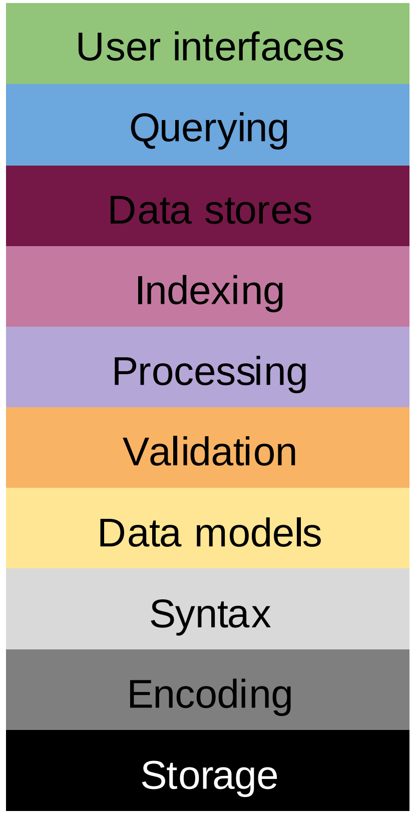
- Big data processing can occur in real-time (stream processing) or in batches (batch processing).

- Processing typically involves distributed systems to handle large volumes of data and complex computations.

Examples:

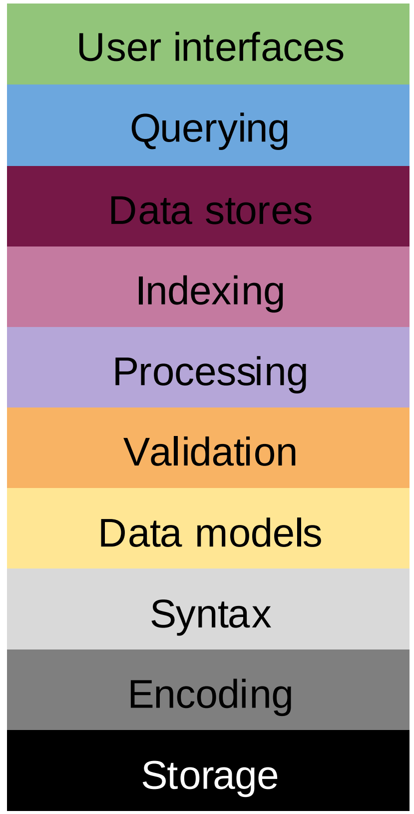
- Batch Processing: Apache Hadoop, Apache Spark

- Stream Processing: Apache Kafka, Apache Flink, Apache Storm



The stack: Validation

- Validation ensures the accuracy, quality, and integrity of the data.
- Before data is analyzed or used in critical applications, it is validated to make sure it conforms to predefined rules and standards.
- Validation is crucial for detecting and correcting errors in big data.



Examples:

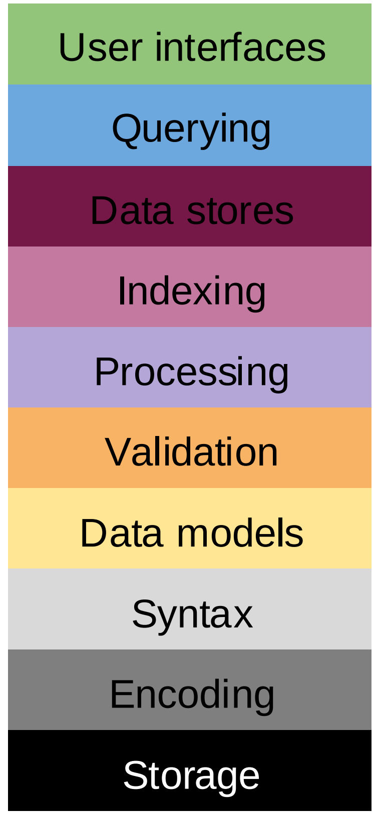
- Schema validation for structured data (e.g., enforcing data types in SQL databases)

- Data cleansing tools to remove duplicates, fix inconsistencies

- Integrity checks in ETL (Extract, Transform, Load) pipelines

The stack: Data Models

- Data models define the structure and relationships of the data
- They describe how data is stored, organized, and interrelated
- Different types of data models are used based on the nature of the data (structured, semi-structured, or unstructured).
- The data model describe how data is stored in the data stores and how it can be queried and processed.



Types of Data Models:

- Relational Data Models: Structured tables, relationships through foreign keys (e.g., SQL databases)
- Document Data Models: JSON-like objects (e.g., MongoDB, CouchDB)
- Graph Data Models: Nodes and edges representing entities and relationships (e.g., Neo4j)

The stack: Syntax

- The syntax of data refers to the rules and structure that define how data is formatted, written, and interpreted.

- It defines the acceptable format for data input, data storage, and data queries.

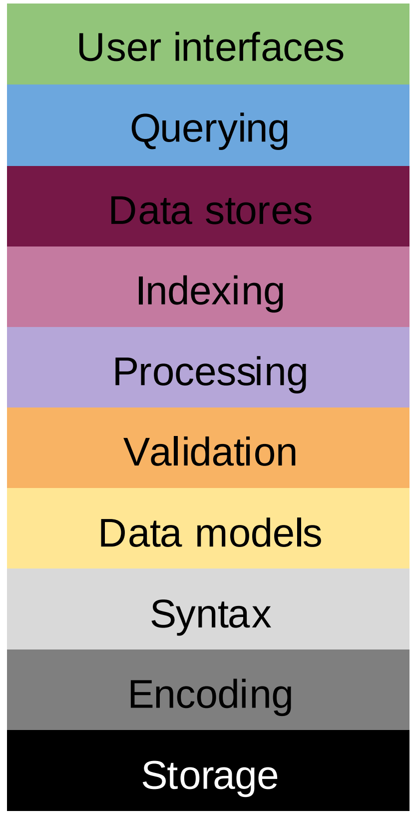
- In big data systems, syntax is critical when writing queries, defining schemas, or encoding data.

Examples:

- SQL Syntax for querying relational databases

- JSON/XML Syntax for semi-structured data

- Regular expression syntax for pattern matching in text data

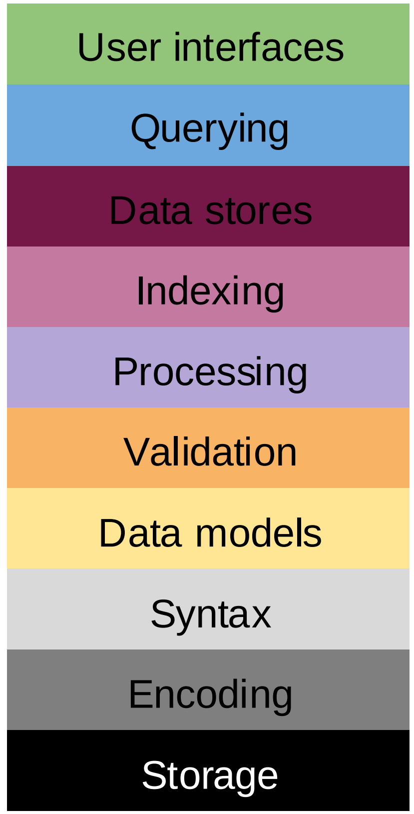


The stack: Encoding

- Encoding refers to the format used to represent data.
- It involves converting data into a specific format that can be efficiently stored, transmitted, or processed.
- Encoding ensures that data is compatible with the system's requirements, secure, and efficiently stored.

Common Encodings:

- Character Encoding: UTF-8, ASCII
- Data Compression: Gzip, Snappy (for efficient storage and transmission)
- Serialization Formats: Avro, Parquet, ORC (commonly used in big data for columnar storage formats)



The stack: Storage

- Storage refers to the physical or cloud-based systems where data is stored and managed.

- In big data environments, storage must be scalable to handle massive amounts of data.

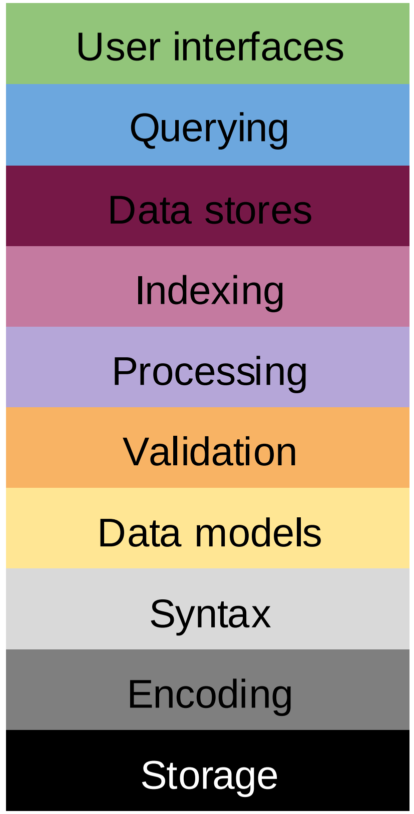
- Depending on the requirements (speed, capacity, reliability), different storage options are used.

Types of Storage:

- Local Storage: Hard drives, SSDs (typically for smaller datasets)

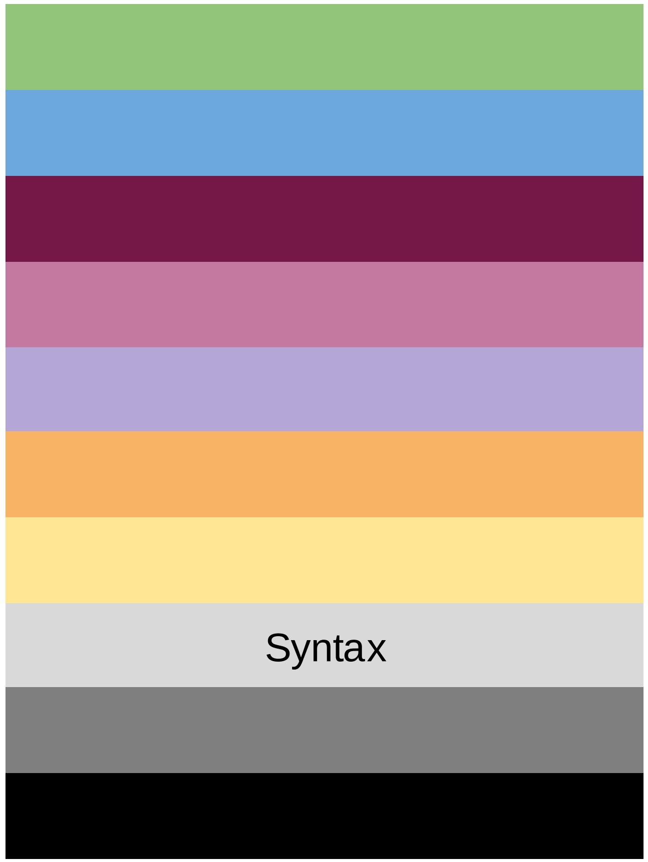
- Distributed Storage Systems: Hadoop HDFS, Amazon S3, Google Cloud Storage

- In-memory Storage: Apache Ignite, Redis (used for faster data access)



The stack: Syntax

- Text
- CSV
- XML
- JSON
- RDF/XML
- Turtle
- XBRL



CSV (Comma separated values)

```
ID,Last name,First name
1,Einstein,Albert
2,Gödel,Kurt
```

ID	Last name	First name
1	Einstein	Albert
2	Gödel	Kurt

RFC 4180

- Specifies the format for (CSV) files. It standardizes the structure of CSV files for more consistent data interchange across systems.

- **Key Elements of RFC 4180:**
    - **1. Field Separation:** Each field in the CSV file is separated by a comma
    - **2. Record Structure:** Each record in the CSV file is a single line of text, and each line contains one or more fields. A record ends with a line break (`CRLF`, represented as `\r\n` in Windows systems).
    - **3. Optional Header Row:**
    - CSV files may have a header row that contains field names. The header is optional but, when present, it is the first line of the file.
    - **4. Quoted Fields:** Fields that contain special characters (such as commas, double quotes, or line breaks) should be enclosed in double quotes (`"`).
    - **5. Empty Fields:** Empty fields are allowed and are represented by two consecutive commas `,,`

Why RFC 4180 is Important

- **Standardization**: Prior to RFC 4180, CSV files had no formal specification, and implementations often varied. This RFC provides a uniform standard that different systems can rely on.

- **Interoperability**: It helps ensure that CSV files can be shared between different applications and environments without formatting issues or data loss.

- This RFC simplifies data exchange and guarantees that developers and systems can follow a consistent set of rules when reading and writing CSV files.

CSV (Comma separated values)

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

RFC 4180
(but lose in practice)

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

- Various applications and systems that generate or read CSV files may not fully adhere to RFC 4180 rules, leading to inconsistent behavior.
- Some CSV files may not quote fields with commas or line breaks correctly.
- Others may use different delimiters (e.g., semicolons) instead of commas.
- In practice, these inconsistencies can cause issues when exchanging CSV files between systems that expect strict adherence to RFC 4180 but encounter non-compliant files.

NORMALIZATION

Normalization can be defined as :

- A process of organizing the data in database to avoid data redundancy, insertion anomaly, update anomaly & deletion anomaly.
- A process of organizing data into tables in such a way that the results of using the database are always unambiguous and as intended.
- Such normalization is intrinsic to relational database theory.
- It may have the effect of duplicating data within the database and often results in the creation of additional tables.

Types of Normalization

- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)
- Boyce-Codd Normal Form (BCNF)
- Fourth Normal Form (4NF)
- Fifth Normal Form (5NF)

First Normal Form (1NF)

First normal form enforces these criteria:

- Eliminate repeating groups in individual tables.
- Create a separate table for each set of related data.
- Identify each set of related data with a primary key

As per the rule of first normal form, an attribute (column) of a table cannot hold multiple values. It should hold only atomic values.

First Normal Form (1NF)

Table_Product		
Product Id	**Colour**	**Price**
1	Black, red	Rs.210
2	Green	Rs.150
3	Red	Rs. 110
4	Green, blue	Rs.260
5	Black	Rs.100

This table is not in first normal form because the "Colour" column contains multiple Values.

After decomposing it into first normal form it looks like:

Product_id	Price
1	Rs.210
2	Rs.150
3	Rs. 110
4	Rs.260
5	Rs.100

Product_id	Colour
1	Black
1	Red
2	Green
3	Red
4	Green
4	Blue
5	Black

Second Normal Form (2NF)

A table is said to be in 2NF if both the following conditions hold:

- Table is in 1NF (First normal form)
- No non-prime attribute is dependent on the proper subset of any candidate key of table.

An attribute that is not part of any candidate key is known as non-prime attribute.

Second Normal Form (2NF)

Table purchase detail		
Customer_id	**Store_id**	**Location**
1	1	Patna
1	3	Noida
2	1	Patna
3	2	Delhi
4	3	Noida

- This table has a composite primary key i.e. customer id, store id.
- The non key attribute is location. In this case location depends on store id, which is part of the primary key.

After decomposing it into second normal form it looks like:

Table Purchase	
Customer_id	**Store_id**
1	1
1	3
2	1
3	2
4	3

Table Store	
Store_id	**Location**
1	Patna
2	Delhi
3	Noida

Third Normal Form (3NF)

A table design is said to be in 3NF if both the following conditions hold:

- Table must be in 2NF
- Transitive functional dependency of non-prime attribute on any super key should be removed.

An attribute that is not part of any candidate key is known as non-prime attribute.

In other words 3NF can be explained like this:

A table is in 3NF if it is in 2NF and for each functional dependency X-> Y at least one

of the following conditions hold:

- X is a super key of table
- Y is a prime attribute of table

An attribute that is a part of one of the candidate keys is known as prime attribute.

Third Normal Form (3NF)

Table Book Details			
Bood_id	**Genre_id**	**Genre type**	**Price**
1	1	Fiction	100
2	2	Sports	110
3	1	Fiction	120
4	3	Travel	130
5	2	Sports	140

In the table, book_id determines genre_id and genre_id determines genre type. Therefore book_id determines genre type via genre_id and we have transitive functional dependency.

A-> B and B->C implies A->C

After decomposing it into third normal form it looks like:

TABLE BOOK		
Book_id	Genre_id	Price
1	1	100
2	2	110
3	1	120
4	3	130
5	2	140

TABLE GENRE	
Genre_id	Genre type
1	Fiction
2	Sports
3	Travel

Boyce-Codd Normal Form (BCNF)

- It is an advance version of 3NF that’s why it is also referred as 3.5NF.
- BCNF is stricter than 3NF.
- A table complies with BCNF if it is in 3NF and for every functional dependency X->Y, X should be the super key of the table.

Boyce-Codd Normal Form (BCNF)

Student	**Course**	**Teacher**
Aman	DBMS	AYUSH
Aditya	DBMS	RAJ
Abhinav	E-COMM	RAHUL
Aman	E-COMM	RAHUL
abhinav	DBMS	RAJ

- KEY: {Student, Course}
- Functional dependency
  {student, course} -> Teacher
  Teacher-> Course
- Problem: teacher is not superkey but determines course.

After decomposing it into Boyce-Codd normal form it looks like:

Student	Course
Aman	DBMS
Aditya	DBMS
Abhinav	E-COMM
Aman	E-COMM
Abhinav	DBMS

Course	Teacher
DBMS	AYUSH
DBMS	RAJ
E-COMM	RAHUL

Fourth Normal Form (4NF)

- Fourth normal form (4NF) is a level of database normalization where there are no non-trivial multivalued dependencies other than a candidate key.

- It builds on the first three normal forms (1NF, 2NF and 3NF) and the Boyce-Codd Normal Form (BCNF).

- It states that, in addition to a database meeting the requirements of BCNF, it must not contain more than one multivalued dependency.

Fourth Normal Form (4NF)

Student	**Major**	**Hobby**
Aman	Management	Football
Aman	Management	Cricket
Raj	Management	Football
Raj	Medical	Football
Ram	Management	Cricket
Aditya	Btech	Football
Abhinav	Btech	Cricket

- Key: {students, major, hobby}
- Multivalued Dependency: students ->-> Major, hobby

After decomposing it into fourth normal form it looks like:

Student	Major
Aman	Management
Raj	Management
Raj	Medical
Ram	Management
Aditya	Btech
Abhinav	Btech

Student	Hobby
Aman	Football
Aman	Cricket
Raj	Football
Ram	Cricket
Aditya	Football
Abhinav	Cricket

Fifth Normal Form (5NF)

A database is said to be in 5NF, if and only if,

- It's in 4NF.
- If we can decompose table further to eliminate redundancy and anomaly, and when we re-join the decomposed tables by means of candidate keys, we should not be losing the original data or any new record set should not arise.
- In simple words, joining two or more decomposed table should not lose records nor create new records.

Fifth Normal Form (5NF)

Seller	Company	Product
Aman	Coca cola company	Thumps Up
Aditya	Unilever	Ponds
Aditya	Unilever	Axe
Aditya	Uniliver	Lakme
Abhinav	P&G	Vicks
Abhinav	Pepsico	Pepsi

- Key: {seller, company, product}
- Multivalued Dependency: Seller ->-> Company, product
- Product is related to company.

After decomposing it into fifth normal form it looks like:

Seller	Product
Aman	Thumps Up
Aditya	Ponds
Aditya	Axe
Aditya	Lakme
Abhinav	Vicks
Abhinav	Pepsi

Seller	Company
Aman	Coca cola company
Aditya	Unilever
Abhinav	P&G
Abhinav	Pepsico

Company	Product
Coca cola company	Thumps Up
Unilever	Ponds
Unilever	Axe
Unilever	Lakme
Pepsico	Pepsi
P&G	Vicks

Data Denormalization (NoSQL)

Boyce-Codd normal form

Normalization

Highly normalized

Highly denormalized

Avoid update anomalies!

Avoid joins!

Anomaly

In the context of databases, anomalies refer to inconsistencies or issues that arise during data operations (such as insertion, deletion, or update) when the database is not properly structured.

These anomalies usually occur in poorly designed databases that have redundant data or are not properly normalized. The most common types of anomalies are:

1. Insertion Anomaly

- It usually happens when certain attributes (columns) are left blank or NULL because the database schema requires data that may not be relevant for all records.

2. Update Anomaly

- Definition: This anomaly occurs when multiple copies of the same data exist, and one or more of them are not updated properly, leading to inconsistent data in the database.
- Example: If a customer's address is stored in multiple places in the database, and the address is updated in only one of the tables but not in the others, this creates inconsistent records.

Anomaly

3. Deletion Anomaly

- Definition: A deletion anomaly occurs when deleting a record unintentionally removes useful information. This is typically a problem in databases with redundant data.
- Example: In a combined "Orders" table, where customer and order information are stored together, if the last order for a customer is deleted, their contact information might also be lost even though we still need to retain the customer’s details.

4. Redundancy and Anomalies

- Redundancy is a common cause of anomalies. When data is duplicated unnecessarily across multiple rows or tables, it increases the chance of anomalies occurring.
- Normalization: One of the key goals of normalization is to reduce redundancy and thus minimize anomalies. By organizing data into multiple related tables and ensuring each piece of data is stored only once, we reduce the risk of inconsistent data updates, problematic inserts, and unintentional deletions.

Denormalized

- Denormalized data refers to a database structure where data redundancy (duplication of data) is allowed for efficiency in query performance, particularly in read-heavy operations.

- In a denormalized structure, data might be stored in a way that combines multiple related pieces of information into fewer tables, reducing the need for complex joins but at the cost of potentially redundant data and update anomalies.

Boyce-Codd normal form

- **Example of denormalized data:**

products	
product	price
varchar(30)	char(1)
Phone	800
Laptop	2000
HDTV	1000
USB Stick	10

sales		
product	customer	quantity
varchar(30)	text	date
Phone	John	1
Phone	Peter	2
Phone	Mary	1
Laptop	John	3
Laptop	Mary	1
HDTV	Mary	2

customers
customer
text
John
Peter
Mary
Bill

Denormalized to first normal form

sales			
product	price	customer	quantity
varchar(30)	char(1)	text	date
Phone	800	John	1
Phone	800	Peter	2
Phone	800	Mary	1
Laptop	2000	John	3
Laptop	2000	Mary	1
HDTV	1000	Mary	2

Collection of tuples

Tuple as a map from strings to values

- A tuple, can be defined as a map from strings to values
- Is a collection of key-value pairs, where each key is a string (representing the column or attribute name), and the corresponding value is the data stored in that attribute.
- Consider the following table:

Student_ID	Name	Age
101	Alice	20

- The corresponding tuple for the row can be thought of as a map from column names (strings) to their respective values:

product $\mapsto$ Phone

price $\mapsto$ 800

customer $\mapsto$ John

quantity $\mapsto$ 1

```
{
  "Student_ID": 101,
  "Name": "Alice",
  "Age": 20
}
```

Syntax of a collection of tuples

sales			
product	**price**	**customer**	**quantity**
varchar(30)	**char(1)**	**text**	**integer**
Phone	800	John	1
Phone	800	Peter	2
Phone	800	Mary	1
Laptop	2000	John	3
Laptop	2000	Mary	1
HDTV	1000	Mary	2

{ "product" : "Phone", "price" : 800, "customer" : "John", "quantity" : 1 }
{ "product" : "Phone", "price" : 800, "customer" : "Peter", "quantity" : 2 }
{ "product" : "Phone", "price" : 800, "customer" : "Mary", "quantity" : 1 }
{ "product" : "Laptop", "price" : 2000, "customer" : "John", "quantity" : 1 }
{ "product" : "Laptop", "price" : 2000, "customer" : "Mary", "quantity" : 1 }
...

Nestedness

- **Nestedness in tuples** refers to the idea that tuples can contain other tuples as elements, creating a hierarchical or nested structure.

- In this way, a tuple doesn't just store simple data values (like integers, strings, etc.), but can also store entire tuples as its values, allowing for the representation of more complex, multi-level data relationships.

<table>
<tr><th colspan="2">sales</th></tr>
<tr><th>product</th><th>orders</th></tr>
<tr><th>varchar(30)</th><th>text</th></tr>
<tr><td>Phone</td><td></td></tr>
<tr><td>Laptop</td><td><table><tr><th>customer</th><th>quantity</th></tr><tr><th>text</th><th>integer</th></tr><tr><td>John</td><td>3</td></tr><tr><td>Mary</td><td>1</td></tr></table></td></tr>
<tr><td>HDTV</td><td><table><tr><th>customer</th><th>quantity</th></tr><tr><th>text</th><th>date</th></tr><tr><td>Mary</td><td>**1**</td></tr></table></td></tr>
</table>

Syntax for nestedness

<table>
<tr><th colspan="2">sales</th></tr>
<tr><th>product</th><th>orders</th></tr>
<tr><th>varchar(30)</th><th>text</th></tr>
<tr><td>Phone</td><td>
<table>
<tr><th>customer</th><th>quantity</th></tr>
<tr><th>text</th><th>integer</th></tr>
<tr><td>John</td><td>1</td></tr>
<tr><td>Peter</td><td>2</td></tr>
<tr><td>Mary</td><td>1</td></tr>
</table>
</td></tr>
</table>

```
{
  "product" : "Phone",
  "orders" : [
    { "customer" : John, "quantity" : 1 },
    { "customer" : Peter, "quantity" : 2 },
    { "customer" : Mary, "quantity" : 2 }
  ]
}
```

The denormalizing road from SQL to NoSQL

A	B	C	D
a	1	alpha	foo
a	2	NULL	foo
b	3	alpha	NULL
b	4	beta	NULL

```
{
  "A" : "a",
  "E" : [
    { "B" : 1, "C" : "alpha" },
    { "B" : 2 }
  ],
  "D" : "foo"
}
```

```
{
  "A" : "b",
  "E" : [
    { "B" : 3, "C" : "alpha" },
    { "B" : 4, "C" : "beta" }
  ]
}
```

Homogeneous collection
of **flat** items.

Relational database

Heterogeneous collection
of **arborescent** items.

Document store

Homogeneous Collection of Flat Items

Homogeneous:
All elements or records in the collection share the same structure. This means that each row in a relational database table has the same set of columns or attributes.

Flat:
The data is organized in a two-dimensional table where each row represents a record, and each column represents an attribute of that record. There is no hierarchy or nesting within the table. Each cell holds a single, atomic value (no lists or sub-objects).

In a relational database, tables are designed with these principles in mind. For example, a table that stores information about books is flat and homogeneous because:
- Each row represents a book.
- Each row has the same columns (Book_ID, Title, Author, Year).
- Each cell holds a single value (not a list, or another object).

Heterogeneous Collection of Arborescent Items (Document Store)

Heterogeneous:
The records in the collection can have varying structures. Unlike a relational database, not every record needs to have the same set of attributes. Different documents (records) can have different fields.

Arborescent:
Refers to tree-like structures (from "arborescence," meaning branching like a tree). In this context, data can have hierarchical or nested structures. Documents can contain nested objects, lists, and complex relationships within a single document.

A document store (such as MongoDB, CouchDB) typically uses a flexible, semi-structured format like JSON or BSON, allowing for varied structures and nesting.

Here:

- The document is arborescent because it includes a nested structure
- The document is heterogeneous because not every document in this collection needs to have the same structure.

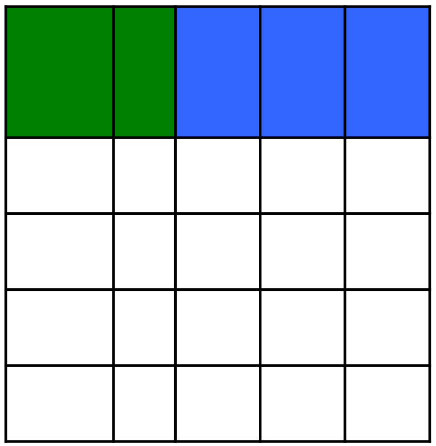
Transition from Homogeneous to Heterogeneous

- The transition from **Homogeneous** to **Heterogeneous** is primarily about moving from a structured, normalized relational approach to a more flexible, often denormalized approach that can better handle complex, hierarchical, or unstructured data.

Semi-Structured Documents

gajus / 123RF Stock Photo

Semi-Structured Documents

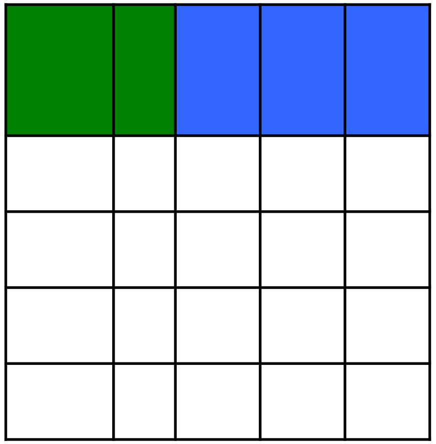


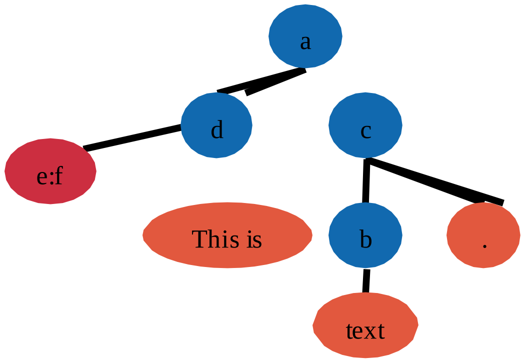
Structured

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Unstructured

Semi-Structured Documents





Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Structured

Semi-structured

Unstructured

Standards

W3C

W3C (World Wide Web Consortium)

- W3C is an international consortium founded in 1994 by Tim Berners-Lee, the creator of the World Wide Web.

- Its purpose is to promote the evolution and growth of the web by ensuring that web technologies are standardized and accessible to all.

- W3C develops a range of standards and recommendations for various aspects of the web, including:
  - **HTML (HyperText Markup Language)**
  - **CSS (Cascading Style Sheets)**
  - **XML (eXtensible Markup Language)**:
  - **SVG (Scalable Vector Graphics)**:
  - **Web Accessibility Initiative (WAI)**: Guidelines for making the web accessible to people with disabilities.

Standards

- **ECMA (European Computer Manufacturers Association)**
- Is an international standardization organization founded in 1961. Its goal is to develop and promote standards for computing and communications.
- Some of the most notable standards developed by ECMA include:
  - **ECMAScript**: This is the standard underlying scripting languages such as JavaScript.
  - **C#**: A programming language developed by Microsoft, the standard of which has been adopted by ECMA.
  - **JSON (JavaScript Object Notation)**: A lightweight data interchange format based on JavaScript object syntax.

- **Key Differences**
- **Scope of application**:
  - W3C primarily focuses on web technologies and accessibility, while ECMA covers a broader range of standards related to computing and communications.

Syntax

Well-formedness

- Well-formedness is a fundamental concept in the context of markup languages, particularly in XML (eXtensible Markup Language), but it can also apply to other structured data formats.
- A document is considered well-formed if it adheres to the syntactical rules defined by the language specification.

Key Characteristics of Well-formedness:

- **Proper Tag Structure**: Every opening tag must have a corresponding closing tag.
- **Nesting Rules**: Tags must be properly nested. This means that an opening tag must be closed before the closing tag of the outer element
- **Attribute Syntax**: Attributes in tags must be properly quoted
- **Root Element**: Document must contain a single root element that encompasses all other elements.
- **Parsing**: Well-formed documents can be correctly parsed by a parsers.

Well-Formed Data in Big Data Systems

In big data systems, well-formed data refers to data that adheres to the expected syntax, structure, or formatting rules, typically defined by schemas or standards.

Steps to Verify Well-Formed Data:

1. Schema and File Format Validation

- **Schema Validation:** Big data systems like Hadoop and Spark rely on data schemas (e.g., Avro, Parquet, JSON Schema).
- Ensure that data conforms to **schema-defined types, constraints, and structure**.
- Big data systems usually enforce schema validation **during read/write operations**.

- **File Format Validation:** Formats like JSON, XML, and CSV must follow specific formatting rules to be considered well-formed:
  - JSON: Check for proper nesting of brackets, quotes, and commas.
  - XML: Ensure matching tags, valid attributes, and proper nesting.
  - CSV: Verify consistent delimiter use and equal number of columns in each row.

- In Python, you can use libraries like json for JSON, xml.etree.ElementTree for XML, or csv for CSV to verify that files are well-formed.

Well-Formed Data in Big Data Systems

2. **Data Cleansing and Validation Tools**:
- Use data cleansing tools like **Trifacta, Talend, or OpenRefine** to analyze and validate large datasets, ensuring the data is complete and well-structured.

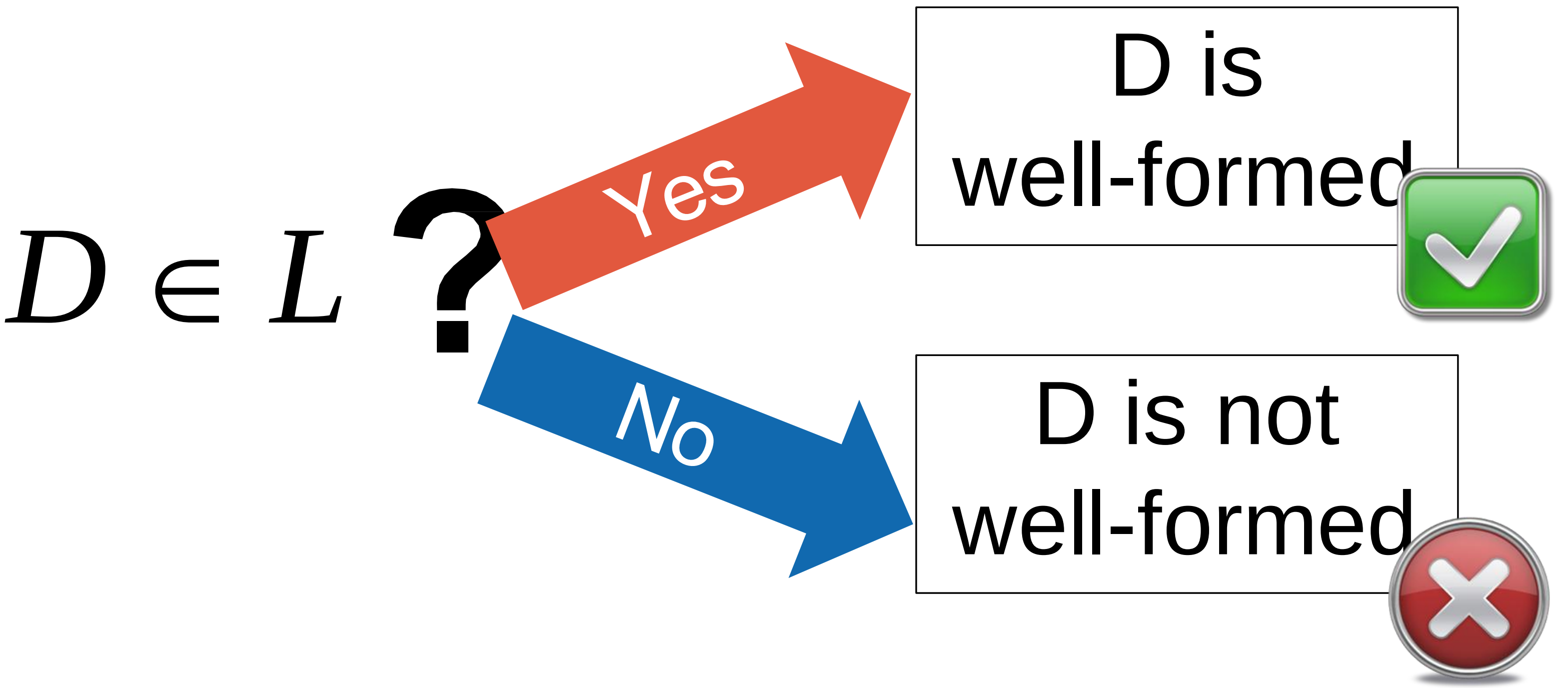
3. **Big Data Frameworks:**
- In systems like **Apache Spark**, you can load data into a DataFrame and run validations against the structure and types of data:

```
from pyspark.sql import SparkSession
 spark = SparkSession.builder.getOrCreate()

 df = spark.read.json("path_to_json_file")
 df.printSchema()  # To check if schema matches expectations
 df.show()  # To see the first few rows of data
```

Well-formedness

One syntax = one language



HTML

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Country</title>
  </head>
  <body>
    <h1 class="Title">Switzerland</h1>
    <div>Population: 8014000<br>
      Currency: Swiss Franc (CHF)</div>
    <h2>Cities</h2>
    <ul>
      <li>Zurich</li>
      <li>Geneva></li>
      <li>Bern <!SS the Federal City SS></li>
    </ul>
  </body>
</html>
```

XML

```
<?xml version="1.0"?>
<country code="CH">
  <name>Switzerland</name>
  <population>8014000</population>
  <currency code="CHF">Swiss Franc</currency>
  <cities>
    <city>Zurich</city>
    <city>Geneva></city>
    <city>Bern <!II the Federal City II></city>
  </cities>
  <description>
    We produce <b>very</b> good chocolate.
  </description>
</country>
```

JSON

```json
{
  "code": "CH",
  "name": "Switzerland",
  "population": 8014000,
  "currency": {
    "name": "Swiss Franc",
    "code": "CHF"
  },
  "confederation": true,
  "president" : "Ueli Maurer",
  "capital": null,
  "cities": [ "Zurich", "Geneva", "Bern" ],
  "description": "We produce very good chocolate."
}
```

JSON

JSON: String

```
"foo"
```

```
"foo\nbar\u005f"
```

JSON: Number

3.1415

-1.2345E+5

JSON: Boolean

```
true
false
```

JSON: Null

`null`

JSON: Array

```
[
  3.14159265368979,
  true,
  "This is a string",
  { "foo" : false },
  null
]
```

JSON: Object

```json
{
  "foo": 3.14159265368979,
  "bar": true,
  "str": "This is a string",
  "obj": { "school" : "ETH"},
  "Q": null
}
```

JSON: Well-formedness

```
{ "foo" : "bar", "foo" : "bar2" }
```

```
{ "foo" : "bar", "bar" : "foo" }
```

(SHOULD)

JSON: Well-formedness

```
{ [ 1 ] : "bar",  2 : "bar2" }
```

```
{ "1" : "bar", "2" : "foo" }
```

JSON: Well-formedness

```
{ foo: "bar",  bar: "bar2" }
```

```
{ "foo" : "bar", "bar" : "foo" }
```

JSON as Data


```json
{
  "target": "Italian",
  "sample": "af0e25c7637fb0dcdc56fac6d49aa55e",
  "choices": ["Chinese", "German", "Italian", "English"],
  "guess": "German",
  "date": "2018-11-07",
  "country": "CH"
}
```

XML

Robert Eastman / 123 Stock Photo

XML: Element

<foo>[more XML]</foo>

XML: Element

<foo>[more XML]</foo>

<bar/> = <bar></bar>

XML: Element

`<foo>[more XML]</foo>`

opening tag

closing tag

`<bar/> = <bar></bar>`

empty tag

XML: Attribute

```
<a attr="value"/>
```

XML: Text

```
<a>This is text</a>
```

XML: Comment

```
<!-- This is a comment -->
```

Text declaration

```
<?xml version="1.0" encoding="UTF-8"?>
```

XML: Well-formedness

```
<?xml version="1.0" encoding="UTF-8"?>
<foo/>
```

XML: Well-formedness

```
<foo/>
```

XML: Well-formedness

```
<?xml version="1.0" encoding="UTF-8"?>
<foo/>
<bar/>
```

XML: Well-formedness

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  <bar/>
</foo>
```

XML: Well-formedness

```
<?xml version="1.0" encoding="UTF-8"?>
text
<foo>
  <bar/>
</foo>
text
```

XML: Well-formedness

```xml
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  text <bar/> text
</foo>
```

XML: Well-formedness

```
<?xml version="1.0" encoding="UTF-8"?>
<foo <element/>>
  <bar></bar>
</foo>
```

XML: Well-formedness

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  text <bar/> text
</foo>
```

XML: Well-formedness

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- this is a comment -->
<foo attribute="<element/>">
  <bar></bar>
  <!-- this is another comment -->
</foo>
<!-- and yet another comment -->
```

XML: Well-formedness

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- this is a comment -->
<foo attribute="element">
  <bar></bar>
  <!-- this is another comment -->
</foo>
<!-- and yet another comment -->
```

XML: Document Type

```
<?xml version="1.0"?>
<!DOCTYPE document>
<document>
Lorem ipsum dolor sit amet, consectetur adipiscing
elit, sed do eiusmod tempor incididunt ut labore et
dolore magna aliqua.
</document>
```

What Appears Where?

	Top-Level	Between Element Tags	Inside Opening Element Tag
Elements	once		
Attributes			
Text			

XML: Well-formedness

```
<a foo="bar" foo="bar2"/>
```

```
<a foo="bar" bar="foo"/>
```

XML: Well-formedness

<a>

<a>

XML: Well-formedness

```
<a>1 < 2</a>
```

```
<a>1 &lt; 2</a>
```

XML: Entity References

```xml
<?xml version "1.0"?>
<document>
2 &lt; 3
</document>
```

XML: Entity References

< → <

XML: Entity References

> ➡ >

XML: Entity References

' ➡ '

XML: Entity References

" → "

XML: Entity References

& → &

XML: Character References (dec)

<?xml version "1.0"?>
<document>
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do

π → π

eiusmod tempor incididunt ut labore et dolore magna aliqua.
</document>

XML: Character References (hex)

<?xml version "1.0"?>
<document>
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do

π → π

eiusmod tempor incididunt ut labore et dolore magna aliqua.
</document>

XML: Well-formedness

```
<a attr="a "quote""/>
```

```
<a attr="a &quot;quote&quot;"/>
```

XML: Well-formedness

```
<a attr="an <element/>"/>
```

```
<a attr="an &lt;element/>"/>
```

XML: Well-formedness

```
<!-- my -- comment -->
```

XML: Document Type

```
<?xml version="1.0"?>
<!DOCTYPE document>
<document>
Lorem ipsum dolor sit amet, consectetur adipiscing
elit, sed do eiusmod tempor incididunt ut labore et
dolore magna aliqua.
</document>
```

XML Names

<1234/>

<a<b/>

<xml/>

<foo1234/>

<_bar/>

ASCII characters allowed in XML names

Control characters															
Control characters															
SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Allowed anywhere in name

Allowed but not at start

not allowed

Whitespaces

XML is W h i t e s p a c e sensitive!

(transmitted to the application using the XML parser)

```
Spaces
Tabs
Carriage return
Newlines
```

Whitespaces

This kind of whitespace
can usually safely
be ignored

(this is very convenient)

```
<items>
  <item>
    <child/>
    <child/>
  </item>
 <item>
    <child/>
    <child/>
  </item>
 <item>
    <child/>
    <child/>
  </item>
</items>
```

Wrap-up

Syntax of a collection of tuples

sales			
product	price	customer	quantity
varchar(30)	char(1)	text	date
Phone	800	John	1
Phone	800	Peter	2
Phone	800	Mary	1
Laptop	2000	John	3
Laptop	2000	Mary	1
HDTV	1000	Mary	2

```
<sale>
  <product>Phone</product>
  <price>800</price>
  <customer>John</customer>
  <quantity>1</quantity>
</sale>
```

Syntax of a collection of tuples

sales			
product	price	customer	quantity
varchar(30)	char(1)	text	date
Phone	800	John	1
Phone	800	Peter	2
Phone	800	Mary	1
Laptop	2000	John	3
Laptop	2000	Mary	1
HDTV	1000	Mary	2

<sales>
<sale><product>Phone</product><price>800</price><customer>John</customer><quantity>1</quantity></sale>
<sale><product>Phone</product><price>800</price><customer>Peter</customer><quantity>2</quantity></sale>
<sale><product>Phone</product><price>800</price><customer>Mary</customer><quantity>1</quantity></sale>
<sale><product>Laptop</product><price>200</price><customer>John</customer><quantity>3</quantity></sale>
...
</sales>

Syntax of a collection of tuples

sales			
product	**price**	**customer**	**quantity**
varchar(30)	**char(1)**	**text**	**date**
Phone	800	John	1
Phone	800	Peter	2
Phone	800	Mary	1
Laptop	2000	John	3
Laptop	2000	Mary	1
HDTV	1000	Mary	2

```xml
<?xml version "1.0"?>
<!DOCTYPE sales>
<sales>
 <sale>
  <product>Phone</product>
  <price>800</price>
  <customer>John</customer>
  <quantity>1</quantity>
 </sale>
 <sale>
  <product>Phone</product>
  <price>800</price>
  <customer>Peter</customer>
  <quantity>2</quantity>
 </sale>
 <sale>
  <product>Phone</product>
  <price>800</price>
  <customer>Mary</customer>
  <quantity>1</quantity>
 </sale>
...
</sales>
```

Nested syntax

<table>
<tr><th colspan="2">sales</th></tr>
<tr><th>product</th><th>orders</th></tr>
<tr><td>varchar(30)</td><td>text</td></tr>
<tr><td>Phone</td><td>
<table>
<tr><th>customer</th><th>quantity</th></tr>
<tr><td>text</td><td>date</td></tr>
<tr><td>John</td><td>1</td></tr>
<tr><td>Peter</td><td>2</td></tr>
<tr><td>Mary</td><td>1</td></tr>
</table>
</td></tr>
</table>

```xml
<?xml version "1.0"?>
<!DOCTYPE sales>
<sales>
 <sale>
  <product>Phone</product>
  <price>800</price>
  <orders>
   <order>
    <customer>John</customer>
    <quantity>1</quantity>
   </order>
   <order>
    <customer>Peter</customer>
    <quantity>2</quantity>
   </order>
   <order>
    <customer>Mary</customer>
    <quantity>1</quantity>
   </order>
  </orders>
 <sale>
...
</sales>
```

Which syntax?

XML
for syntax

JSON
2021
for syntax

YAML (the Python-like JSON)

```yaml
%YAML 1.2
---
Country:
  code: 'CH'
  name: 'Switzerland'
  population: 8014000
  currency:
    name: 'Swiss Franc'
    code: 'CHF'
  confederation: true
  president : Ueli Maurer'
  capital: null
  cities:
    - 'Zurich'
    - 'Geneva'
    - 'Bern'
  description: 'We produce very good chocolate.'
```

NIfTI (Neuroimaging Informatics Technology Initiative)

NIfTI is a file formats that is essential for handling and processing large datasets of medical images.

- **Purpose**: NIfTI is primarily used for storing neuroimaging data, such as MRI (Magnetic Resonance Imaging) and fMRI (functional MRI) scans.
- **Structure**: The NIfTI format is efficient for storing 3D and 4D medical imaging data, making it highly suitable for neuroimaging studies that generate large volumes of data over time. It supports both single-file formats (`.nii`) and dual-file formats (`.hdr` and `.img`).

- **Advantages for Big Data:**
- **Compact and Standardized**: NIfTI is designed to efficiently handle and store complex data, and its compactness is crucial when processing large data.
- **Metadata Support**: It contains spatial metadata, such as voxel size and orientation, which is critical when analyzing large datasets in neuroscience.
- **Ease of Use in Machine Learning**: NIfTI is commonly used in machine learning and AI models for medical image analysis due to its simplicity and compatibility

DICOM (Digital Imaging and Communications in Medicine)

- **Purpose**: DICOM is the universal standard for transmitting, storing, and sharing medical imaging information, such as CT, MRI, and X-ray images. It is widely used in hospitals and medical facilities.
- **Structure**: DICOM files contain not only the image data but also extensive metadata, such as patient information, scan settings, and device information. These files have both image and non-image components.

- **Advantages for Big Data:**
- **Comprehensive Data:** DICOM encapsulates both imaging data and detailed metadata, providing a complete set of information. This is critical for large-scale healthcare applications where patient data must be traceable.
- **Interoperability:** DICOM's standardization ensures that images can be shared and interpreted across different medical devices and systems globally. This is essential when aggregating big data from multiple sources.
- **PACS Integration:** DICOM is integral to PACS (Picture Archiving and Communication Systems), which is used to manage large volumes of medical images across health systems.

Differences in Big Data Context

- NIfTI is more focused on neuroimaging and research settings, where large 3D/4D datasets are processed for brain imaging and analysis. It is lightweight and often used in AI applications for image analysis in neuroscience.

- DICOM, on the other hand, is used broadly across all types of medical imaging, offering robust metadata support and integration with clinical workflows. It plays a vital role in hospital and healthcare settings, particularly in the storage, retrieval, and sharing of large image datasets across departments.