ARTICLE 9

HTML, XML, and JSON

Abhishek Sharma
Loyola University Chicago

Nenad Jukić
Loyola University Chicago

In this article, we will give a brief overview of HTML, XML, and JSON, and describe how they relate to database systems.

Markup Languages

A **markup language** is a language to annotate (i.e., to "mark up") text in a document, where annotations add functionalities to the text. Depending on the type of markup language, the annotations can be used for various purposes, such as indicating how the annotated text in the document is to be formatted, presented, structured, and so on.

Hypertext markup language (HTML) is an example of a markup language. HTML provides functionality for describing how to display the information contained in web pages. The purpose of HTML is strictly to facilitate the display of data.

Consider the HTML code shown in Figure 1, which is stored in a file titled *HTMLExample.html*. The code contains text annotated by the HTML tags. For example, the text annotated by the start tag and end tag is marked up as text to be displayed as a separate paragraph, and the text annotated by the start tag and end tag is annotated as text to be to be displayed in bold font.

```
<html>
<head>
<title> Web page EXAMPLE</title>
</head>
<body>
<P>This is an example of a HTML web page.</P>
<B><P>This part is in bold font.</P></B>
<I><P>This part is in italic font.</P></I>
</body>
</html>
```

Figure 1 An example of HTML code.

HTMLExample.html, containing the HTML code shown in Figure 1, will be retrieved by a web browser as the web page shown in Figure 2.

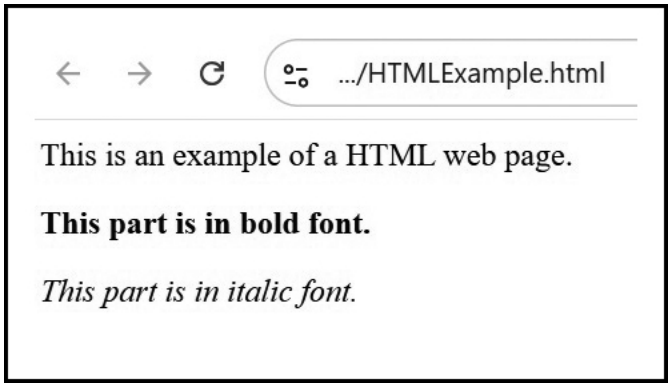

Figure 2 An example of a web page.

XML

Extensible markup language (XML) is a markup language for adding structure and semantics to a document. One of the common usages of XML is to facilitate data exchange from databases in e-business applications. In such scenarios, data is extracted from databases, formatted as XML documents, and transported to and displayed on web pages using HTML. The following is a brief overview of some of the most basic functionalities of XML.

The basic building block of an XML document is called an "element." An XML document is composed of the "root element," which contains all other elements in the document. Elements in an XML document are identified by a start tag <element_name> and an end tag </element_name>. "Simple elements" are the actual data values (e.g., an integer or a character value). "Complex elements" are elements that contain other elements.

Consider the XML code shown in Figure 3. The first line of this code is the XML declaration.* The rest of this XML code is based on a subset of data in the HAFH database (used in Chapters 3 and 5 of *Database Systems: Introduction to Databases and Data Warehouses [Edition 3.0]*.) that depicts buildings, apartments, and corporate clients.

```
<?xml version="1.0" ?>
<HAFH>
<Building>
<BuildingID>B1</BuildingID>
<BNoOfFloors>5</BNoOfFloors>
<Apartment>
<AptNo>21</AptNo>
<ANoOfBedrooms>1</ANoOfBedrooms>
</Apartment>
<Apartment>
<AptNo>41</AptNo>
<ANoOfBedrooms>1</ANoOfBedrooms>
<CorporateClient>
<CCID>C111</CCID>
<CCName>BlingNotes</CCName>
</CorporateClient>
</Apartment>
</Building>
<Building>
<BuildingID>B2</BuildingID>
<BNoOfFloors>6</BNoOfFloors>
. . .
. . .
</HAFH>
```

Figure 3 An example of XML code.

In this simple example, the root of the tree is the HAFH complex element that contains Building elements. Each Building element is a complex element that contains two simple elements—*BuildingID* and *BNoOfFloors*—and one or more *Apartment* elements. Each *Apartment* element is a complex element that has two simple elements—*AptNo* and *ANoOfBedrooms*. In addition, some *Apartment* elements contain *CorporateClient* elements. Each CorporateClient element is a complex element that has two simple elements—*CCID* and *CCName*.

In XML all elements are hierarchically organized. The terms "parent element," "child element," "descendant element," and "sibling element" are used to describe the hierarchical relationship between the elements. For example, *Building* is a parent element of child elements *BuildingID*, *BNoOfFloors*, and *Apartment*, while *Apartment* is a parent element of child elements *AptNo*, *ANoOfBedrooms*, and *CorporateClient*. At the same time, *BuildingID*, *BNoOfFloors*, *Apartment*, *AptNo*, *ANoOfBedrooms*, *CorporateClient*, *CCID*, and *CCName* are all descendant elements of *Building*. Sibling elements are elements that share the same parent. For example, *BuildingID* and *BNoOfFloors* are sibling elements.

Since the XML data is hierarchical, the schema of XML data is described by a tree. Figure 4 depicts the schema of the *HAFHInfo.xml* document shown in Figure 3.

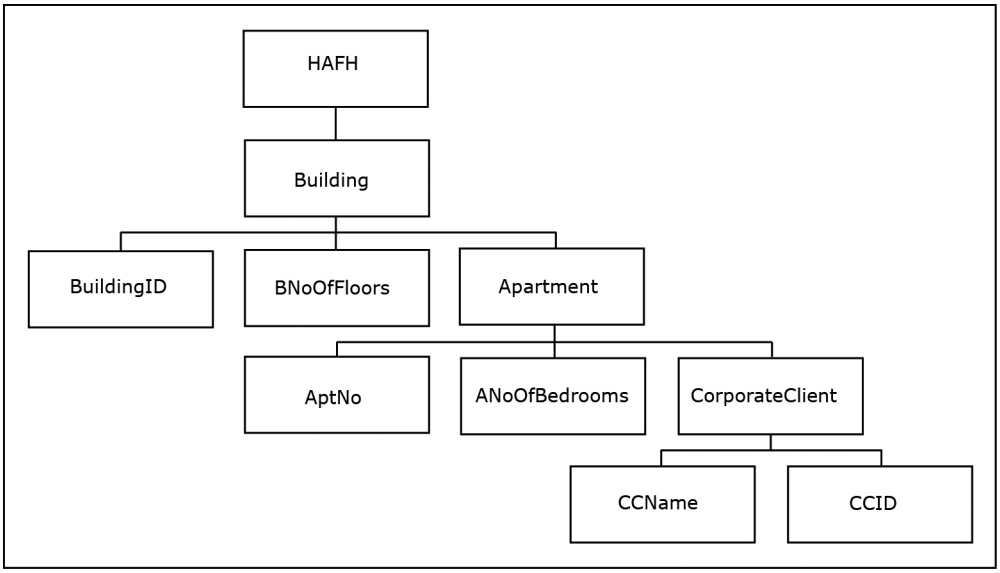

Figure 4 The schema of the XML data shown in Figure 3.

* The XML declaration identifies the document as being XML and specifies the version of the XML standard (in this case, 1.0).

As illustrated by Figure 3, the data extracted from a database can be captured in an XML document. In addition, an XML document itself can be stored in a database. There are different ways to store XML documents in a database. XML documents can be stored as text in a database system, which requires the DBMS to have a module for document processing. Alternatively, the XML document contents can be stored as data elements, if all the documents have the same structure. In such cases, the XML schemas must then be mapped to a database schema. It is also possible to create XML documents from a preexisting relational database and store them back into a database. Special DBMS called "native XML databases" have been designed specifically for storing and processing XML data. Native XML databases have an XML document as the basic unit of storage. Many DBMS have built-in functions that will present the data in XML elements in a typical relational format, as rows of a table.

XML Queries

XML is often used for exchanging data between a database and an application. In such cases, XML documents can contain fairly complex structures of elements. Thus, it is necessary to have a mechanism for traversing the tree structure of XML elements and for extracting information contained within.

XML path language (XPath) is a simple language that utilizes the tree representation of an XML document, allowing the user to travel around the tree. XPath expressions return a collection of element nodes from the tree that satisfy patterns specified in an expression. Separators / and // are used to specify the children or descendants of a tag as described by the following:

```
/          means the tag must appear as the immediate descendant
           (e.g., child) of a previous parent tag
//         means the tag can appear as a descendant of a previous tag
           at any level
```

For example, to access the buildings in the HAFH database, the following statement would be issued:

```
/HAFH/building
```

As another example, consider the following statement that accesses the apartment numbers of apartments with more than one bedroom:

```
//apartment [ANoOfBedrooms > 1]/AptNo
```

XQuery is a language used for more wide-ranging queries on XML documents. XQuery provides functionalities for querying XML documents similar to SQL functionalities for querying relational tables. XQuery uses XPath expressions within a "FLWOR" (*For, Let, Where, Order by, Return*) expression, which is similar to the SELECT . . . FROM . . . WHERE expressions of SQL. The *For* part extracts elements from an XML document. The *Let* part allows the creation of a variable and the assignment of a value to it. The *Where* part filters the elements based on a logical expression. The *Order* by part sorts the result. The *Return* part specifies what is to be returned as a result. The *For* and *Let* in the FLWOR expression can appear any number of times or in any order. The *Let*, *Where*, and *Order by* are optional, while the *For* and *Return* are always needed. To illustrate these expressions, consider the following example, which assumes that the XML document shown in Figure 3 is saved into a document titled *hafhinfo.xml* and stored on the web server www.hafhrealty.com:

```
FOR $x in doc(www.hafhrealty.com/hafhinfo.xml)
RETURN <res> $x/CorpClient/ccname, $x/CorpClient/ccid </res>
```

This XQuery is requesting a list of the names and IDs of corporate clients. Now consider the expanded example, requesting a list of the names and IDs of corporate clients who are renting apartments that have more than one bedroom, sorted by the IDs of corporate clients:

```
FOR $x in doc(www.hafh.com/hafhinfo.xml) LET $minbedroms := 2
WHERE $x/ANoOfBedrooms >= $minbedroms
ORDER BY $x/CorpClient/ccid
RETURN <res> $x/CorpClient/ccname, $x/CorpClient/ccid </res>
```

Apart from having efficient mechanisms for extracting specific elements and data from an XML document, there is also the need for efficient means for automated construction of XML documents, based on the data queried from a database. Many of the dominant database management systems today incorporate XML by including **SQL/XML**, which is an extension of SQL that specifies the combined use of SQL and XML. The following is a simple example that illustrates one of the particular functionalities of SQL/XML.

Consider SQL Query A, which retrieves the content of the table INSPECTOR in the HAFH database.

SQL Query A:

```
SELECT      i.insid, i.insname
FROM        inspector i;
```

SQL Query A results in the following output.

SQL Query A Result:

```
Insid       insname
I11         Jane
I22         Niko
I33         Mick
```

SQL/XML Query AX utilizes SQL/XML function xmlelement() to create an XML element.

SQL/XML Query AX:

```
SELECT      xmlelement(name "inspector",
   xmlelement(name "insid", i.insid),
   xmlelement(name "insname", i.insname) )
FROM        inspector i;
```

SQL/XML Query AX results in the following output.

SQL/XML Query AX Result:

```
<inspector>
   <insid>I11</insid>
   <insname>Jane</insname>
</inspector>
<inspector>
   <insid>I22</insid>
   <insname>Niko</insname>
</inspector>
<inspector>
   <insid>I33</insid>
   <insname>Mick</insname>
</inspector>
```

Whereas the SQL Query A produced rows and columns of data from the table INSPECTOR, the SQL/XML Query AX produced XML elements based on the data in the table INSPECTOR.

One of the typical uses of XML is as a means for presenting database content on web pages. One of the ways to present the XML elements retrieved from the database on a web page is by using **Extensible Stylesheet Language (XSL)**.

Consider the XML file *inspectors.xml* shown in Figure 5. The first line of code is the XML declaration. The second line of code specifies that an XSL

```
<?xml version="1.0" ?>
<?xml-stylesheet type="text/xsl" href="insptoweb1.xsl"?>
<buildinginspectors>
<inspector>
<insid>I11</insid>
<insname>Jane</insname>
</inspector>
<inspector>
<insid>I22</insid>
<insname>Niko</insname>
</inspector>
<inspector>
<insid>I33</insid>
<insname>Mick</insname>
</inspector>
</buildinginspectors>
```

Figure 5 The XML file inspectors.xml (to be displayed as a web page).

file *insptoweb1.xsl* (shown in Figure 6) will be used to describe how the file *inspector.xml* should be displayed as a web page. The remainder of the code in the XML file inspectors.xml contains the content from the table INSPECTOR retrieved by *Query AX*.

The XSL file *insptoweb1.xsl* shown in Figure 6 describes how the file *inspector.xml* will be displayed as a web page.

The first four lines and the last two lines of the code in Figure 6 contain the necessary utilitarian information for XSL documents. The tags <html>, <title>, <body>, <b>, and <p> used after the first four lines of the code are standard HTML tags illustrated by the example shown in Figures 1 and 2. The code in the center, shown in bold font, uses the XSL element <xsl:for-each> to select (in the XML file referencing this XSL file) every XML tree element identified by the XPath expression buildinginspectors/inspector. When the XML file *inspectors.xml* is opened by a web browser, it is displayed as shown in Figure 7.

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <head>
  <title> Web page XML EXAMPLE1</title>
  </head>
  <body>
    <B>Building Inspectors</B>
    <xsl:for-each select="buildinginspectors/inspector">
   <P> --- </P>
   <P>Inspector ID: <xsl:value-of select="insid" /></P>
   <P>Inspector Name: <xsl:value-of select="insname"
/></P>
    </xsl:for-each>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

Figure 6 The XSL file *insptoweb1.xsl* (formats *inspectors.xml* as a web page).

To illustrate how the same XML elements can be formatted differently, Figure 8 shows the file *insptoweb2.xsl*, which contains a different version of the XSL code for the file *inspectors.xml*. This code uses HTML tags for coloring and table formatting. For example, tag `<tr bgcolor="#808080">` specifies the use of a gray background color in the header of the table.

Assume that the XML file *inspectors.xml* now refers to the XSL file *insptoweb2.xsl* instead of the XSL file *insptoweb1.xsl*. In that case, the XML file *inspectors.xml* will be displayed as shown in Figure 9 when opened by a web browser.

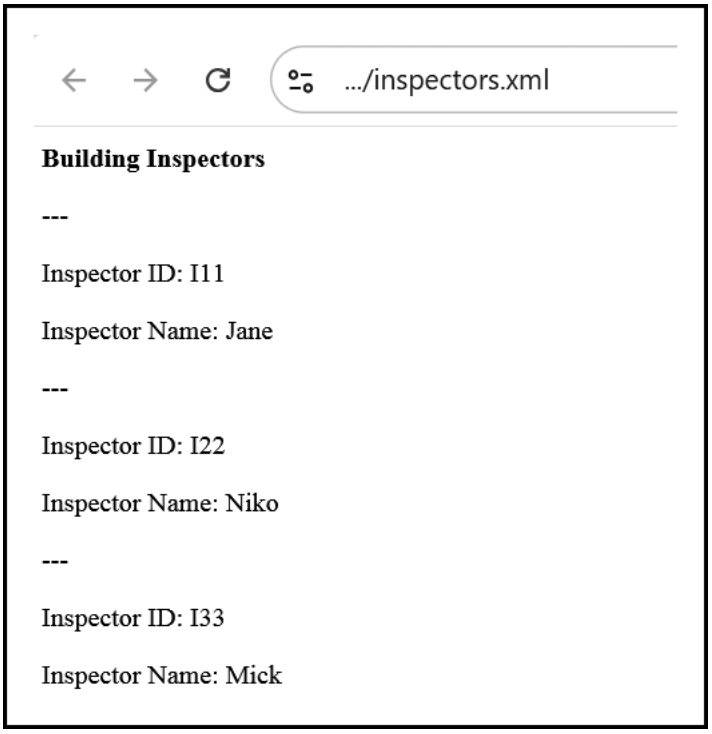

Figure 7 The XML file *inspectors.xml* (displayed as a web page).

As we just illustrated, the extracted database content annotated by the XML tags in an XML document can be transported to various destinations and arranged, formatted, and presented in various ways.

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <head>
  <title> Web page XML EXAMPLE2</title>
  </head>
  <body>
    <B>Building Inspectors</B>
    <table border="1">
<tr bgcolor="#BBFFFF">
   <th>Inspector ID</th>
   <th>Inspector Name</th>
    </tr>
    <xsl:for-each select="buildinginspectors/inspector">
    <tr>
   <td><xsl:value-of select="insid" /></td>
   <td><xsl:value-of select="insname" /></td>
    </tr>
    </xsl:for-each>
    </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

Figure 8 The XSL file *insptoweb2.xsl* (formats inspectors.xml as a web page).

JSON: A Lightweight Data Interchange Format

JavaScript Object Notation (JSON) is a popular data format that is easy to read and write, and convenient to parse and generate for various front-end applications (including application programming interfaces—APIs). Unlike HTML and XML, JSON is not a markup language, but a data interchange format. In markup languages, data is represented by tags, and in JSON, data is represented by less verbose key-value pairs. JSON is derived from JavaScript but is language-independent and is mostly commonly used for transmitting data between a server and a web application. In recent years, JSON has become a dominant format for APIs and web services, and this popularity is mainly driven by its simplicity and ease of use. It is also called a lightweight data interchange format because it has reduced overhead, and it is less complex compared to other data formats.

JSON can represent all the functionality of various architectures of a traditional relational database. JSON supports several data types, including

String (e.g., "Tina")
Number (e.g., 47, 3.14)
Boolean (true or false)
Null (null)
Array (e.g., ["Easy Boot", "Cosy Sock", "Dura Boot"])
Object (a nested JSON structure with key-value pairs)

As shown earlier, the data for SQL Query A in the context of XML will be shown as SQL/XML Query AX Result. The equivalent data about inspectors could be represented by the following JSON object:

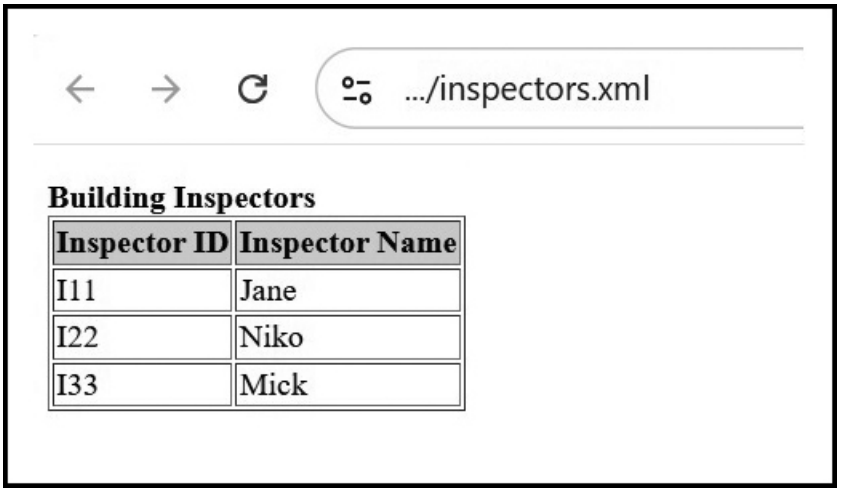

Figure 9 The XML file *inspectors.xml* (displayed as another web page).

Data from SQL Query AX Represented as a JSON Object:

```
{
        "Inspectors": [
          {
            "Insid": "I11",
            "Insname": "Jane"
          },
          {
            "Insid": "I22",
            "Insname": "Niko"
          },
          {
            "Insid": "I33",
            "Insname": "Mick"
          }
        ]
}
```

The "[" in the JSON structure represents the start of an array. In JSON, an array is a list of items, which can include objects, strings, numbers, Booleans, or other arrays. In the example above, key-value pairs, where the key is on the left side of a colon and the value is on the right side of a colon (such as key-value pair "Insid": "I22" and key-value pair "Insname": "Niko"), form objects (delimited by curly braces) which are grouped in an array. This entire array is now a pair part of a key-value pair, where the key part is "Inspectors". In other words, the value of the Inspectors key is an array (indicated by [. . .]). This array contains objects (indicated by {. . .}), and each object represents an inspector with properties such as "Insid" and "Insname". This structure is particularly useful when we have a collection of similar data, like a list of inspectors, products, or transactions. It allows easy access and manipulation of each item in the array programmatically.

Unlike relational databases, data in JSON format could be architected in multiple ways to match a very specific way of querying or accessing the data that is common in online transaction processing. If data is used in a manner where tables need not be joined, and each table (store, region, transaction, product, etc.) will be used independently, the JSON data could be represented in a "flat" manner, as shown in Figure 10.

```
{
    "Stores": [
   { "StoreID": "S1", "StoreZip": 60600, "RegionID": "C" },
......
    ],
    "Regions": [
   { "RegionID": "C", "RegionName": "Chicagoland" },
......
    ],
    "Transactions": [
   { "TID": "T111", "CustomerID": "1-2-333", "StoreID": "S1" },
......
    ],
    "Products": [
   { "ProductID": "1X1", "ProductName": "Zzz Bag", "VendorID": "PG" },
......
    ]
}
```

Figure 10 JSON object—Example 1.

However, this data representation will complicate the query on JSON if data needs to be joined across various tables/entities like transaction and product. If the data is often used by joining multiple tables, then the JSON format could be changed accordingly to dramatically enhance query performance. For example, if regions are accessed along with details of stores in them or transactions are accessed often with products' information, the JSON data could be arranged in a "nested" manner, as shown in Figure 11.

```
{
    "Regions": [
   {
  "RegionID": "C",
  "RegionName": "Chicagoland",
  "Stores": [
    { "StoreID": "S1", "StoreZip": 60600 },
......
  ]
   }
    ],
    "Transactions": [
   {
  "TID": "T111",
  "StoreID": "S1",
  "Products": [
    { "ProductID": "1X1", "ProductName": "Zzz Bag" },
......
  ]
   }
    ]
}
```

Figure 11 JSON object—Example 2.

In this example, store data is nested in the regional data and product data is nested in the transactions data. If data is accessed by region, the query will be very efficient. Similarly, product data is nested in transactions data, and if a query often accesses transactions along with product information, it will be significantly better than previous representation of data.

Nesting data in JSON could be arranged to various degrees to suit the application's needs. For example, if region data, store data, transaction data, and product data are accessed together (i.e., data about the products sold within transactions that occur in stores located in regions), all four entities could be nested under the region. This is shown in Figure 12.

```
{
    "Regions": [
   {
  "RegionID": "C",
  "RegionName": "Chicagoland",
  "Stores": [
    {
   "StoreID": "S1",
   "StoreZip": 60600,
   "Transactions": [
       {
                            "TID": "T111",
                            "Products": [
                           {
                                              "ProductID": "1X1",
                                              "ProductName": "Zzz
Bag"
                           }
                            ]
       }
   ]
    }
  ]
   }
    ]
}
```

Figure 12 JSON object—Example 3.

The flexibility of JSON is that you can arrange the nesting and resulting architecture according to how data will be accessed. As another example, if data is queried by customers, we can nest transactions and products under customers, as shown in Figure 13.

```
{
    "Customers": [
   {
  "CustomerID": "1-2-333",
  "CustomerName": "Tina",
  "Transactions": [
    {
   "TID": "T111",
   "StoreID": "S1",
   "Products": [
       { "ProductID": "1X1", "ProductName": "Zzz Bag" }
   ]
    }
  ]
   }
    ]
}
```

Figure 13 JSON object—Example 4.

The key factors in choosing JSON architecture and respective nesting strategies are the following:

1. **Query performance:** If we frequently query data grouped by a specific entity (e.g., regions or customers), nesting related data under that entity reduces lookup time.
2. **Size of the resulting JSON data:** Deeply nested JSON can increase size and complexity, especially if there is redundancy. Hence query performance comes at a cost.
3. **Flexibility:** Flat structures are easier to work with if we need relational operations or transformations.
4. **Requirements of the front-end application:** API endpoints or front-end data requirements significantly influence a particular structure.

Overall, JSON is becoming a popular format in today's data world and many relational databases (PostgreSQL, Snowflake, and more) have adopted JSON as a data type. Data stored in JSON fields could be queried using SQL extensions inside these databases.

As an example, we can create a product table with a JSON column in PostgreSQL database using the following syntax:

```
CREATE TABLE product (
    product_id SERIAL PRIMARY KEY,
    product_data JSONB
);
-- Insert JSON data into the table
INSERT INTO product (product_data)
VALUES
    ('{"ProductID": "1X1", "ProductName": "Zzz Bag", "ProductPrice": 100, "Category":
            {"CategoryID": "CP", "CategoryName": "Camping"}}'),
    ('{"ProductID": "2X2", "ProductName": "Easy Boot", "ProductPrice": 70, "Category":
            {"CategoryID": "FW", "CategoryName": "Footwear"}}'),
    ('{"ProductID": "3X3", "ProductName": "Cosy Sock", "ProductPrice": 15, "Category":
            {"CategoryID": "FW", "CategoryName": "Footwear"}}'
......
   );
```

If we wish to query the entire JSON object, we can query the column like a regular table column in the following manner:

```
SELECT product_data
FROM product;
```

In addition to this, we can also query the JSON object (by using ->) and also retrieve a JSON value (by using ->>). For example, if we wish to display the product name for all products, we would write the following query:

```
SELECT product_data ->> 'ProductName' AS ProductName
FROM product;
```

Using both -> and ->>, we can query the nested JSON object. For example, if we wish to display category names for all products, we would write the following query:

```
SELECT product_data -> 'Category' ->> 'CategoryName' AS CategoryName FROM product;
```

In this example, the -> syntax gives us the array of category, and then the subsequent ->> gives us the category names' values.

With these latest developments, JSON is becoming a widely used data paradigm and has been gaining strong footing even in the traditional relational database world.

There are salient differences between JSON and XML that have led to the emergence of JSON as the preferred way to exchange data across applications. The big shortcoming of XML is that it is highly verbose and repeats information across the entire document. In the example shown earlier (SQL/XML Query AX Result), the phrase "inspector" is repeated multiple times and does not bring any new information. However, a matching representation

of the same information in JSON (Data from SQL Query AX Represented as a JSON Object) does not repeat the phrase "inspector." Even though JSON is not as compact as a relational database, it is much less verbose than XML.

Another issue with XML is that even though XML and JSON both are hierarchical in structure, XML is not optimal for quick querying of the data and retrieving a particular set of information. Similar information stored in JSON is very efficiently retrieved by existing technology and tools like Python, JavaScript, PostgreSQL, Microsoft PowerShell, Linux Bash, and so on. With JavaScript, which is a language very often used in mobile or web applications, the difference is notable. A single command looking for the name of the inspector I22 listed below will retrieve the inspector named "Niko" into the variable inspector and the subsequent command will display "Niko" as the inspector's name:

```
// Retrieve the Inspector node
const inspector = inspectors.Inspectors.find(inspector => inspector.Insid === "I22");
```

A similar code for XML, as shown here, will be more verbose and inefficient:

```
// Retrieve the Inspector node
const inspectors = xmlDoc.getElementsByTagName("Inspector");
for (let i = 0; i < inspectors.length; i++) {
    if (inspectors[i].getElementsByTagName("Insid")[0].textContent === "I22")
}
```

Another reason why JSON has become more prevalent in recent years is the flexibility of JSON (as discussed earlier) with different levels of nesting of the same data to suit a particular application. However, with XML, a singular hierarchical format is strictly imposed, and there is no way around it.

Overall, JSON has been adopted by most applications and programming paradigms today mainly because it provides less overhead, efficient parsing and consumption of data by applications, and flexibility of architecture to match the needs of the application. With that said, there are still use cases where XML is a better choice. For example, when dealing with configuration files supporting applications or enterprise systems where verbosity of the data is required, XML is still widely used.