# [XML](https://docs.microsoft.com/en-us/sql/relational-databases/xml/xml-data-sql-server)

## XML Data

SQL Server provides a powerful platform for developing rich applications for semi-structured data management. Support for XML is integrated into all the components in SQL Server and includes the following:

* The **xml** data type. XML values can be stored natively in an **xml** data type column that can be typed according to a collection of XML schemas, or left untyped. You can index the XML column.
* The ability to specify an XQuery query against XML data stored in columns and variables of the **xml** type.
* Enhancements to OPENROWSET to allow bulk loading of XML data.
* The FOR XML clause, to retrieve relational data in XML format.
* The OPENXML function, to retrieve XML data in relational format.

## XML Data Type and Columns

### Relational or XML Data Model

If your data is highly structured with known schema, the relational model is likely to work best for data storage. On the other hand, if the structure is semi-structured or unstructured, or unknown, you have to give consideration to modeling such data.

XML is a good choice if you want a platform-independent model in order to ensure portability of the data by using structural and semantic markup. Additionally, it is an appropriate option if some of the following properties are satisfied:

* Your data is sparse or you do not know the structure of the data, or the structure of your data may change significantly in the future.
* Your data represents containment hierarchy, instead of references among entities, and may be recursive.
* Order is inherent in your data.
* You want to query into the data or update parts of it, based on its structure.

### Reasons for Storing XML Data in SQL Server

Following are some of the reasons to use native XML features in SQL Server instead of managing your XML data in the file system:

* You want to share, query, and modify your XML data in an efficient and transacted way. Fine-grained data access is important to your application. For example, you may want to extract some of the sections within an XML document, or you may want to insert a new section without replacing your whole document.
* You have relational data and XML data and you want interoperability between both relational and XML data within your application.
* You need language support for query and data modification for cross-domain applications.
* You want the server to guarantee that the data is well formed and also optionally validate your data according to XML schemas.
* You want indexing of XML data for efficient query processing and good scalability, and the use of a first-rate query optimizer.
* You want SOAP, ADO.NET, and OLE DB access to XML data.
* You want to use administrative functionality of the database server for managing your XML data. For example, this would be backup, recovery, and replication.

If none of these conditions is satisfied, it may be better to store your data as a non-XML, large object type, such as **[n]varchar(max)** or **varbinary(max)**.

### XML Storage Options

The storage options for XML in SQL Server include the following:

* Native storage as **xml** data type

The data is stored in an internal representation that preserves the XML content of the data. This internal representation includes information about the containment hierarchy, document order, and element and attribute values.

* Mapping between XML and relational storage

By using an annotated schema (AXSD), the XML is decomposed into columns in one or more tables. This preserves fidelity of the data at the relational level.

* Large object storage, **[n]varchar(max)** and **varbinary(max)**

An identical copy of the data is stored. This is useful for special-purpose applications such as legal documents. Most applications do not require an exact copy and are satisfied with the XML content (InfoSet fidelity).

#### Choice of XML Technology

The choice of XML technology, native XML versus XML view, generally depends upon the following factors:

* Storage options

Your XML data may be more appropriate for large object storage (for example, a product manual), or more amenable to storage in relational columns (for example, a line item converted to XML). Each storage option preserves document fidelity to a different extent.

* Query capabilities

You may find one storage option more appropriate than another, based on the nature of your queries and on the extent to which you query your XML data. Fine-grained query of your XML data.

* Indexing XML data

You may want to index the XML data to speed up XML query performance. Indexing options vary with the storage options;

* Data modification capabilities

Some workloads involve fine-grained modification of XML data. For example, this can include adding a new section within a document, while other workloads, such as Web content, do not. Data modification language support may be important for your application.

* Schema support

Your XML data may be described by a schema that may or may not be an XML schema document. The support for schema-bound XML depends upon the XML technology.

Different choices also have different performance characteristics.

### Granularity of XML Data

The granularity of the XML data stored in an XML column is very important for locking and, to a lesser degree, it is also important for updates. SQL Server uses the same locking mechanism for both XML and non-XML data. Therefore, row-level locking causes all XML instances in the row to be locked. When the granularity is large, locking large XML instances for updates causes throughput to decline in a multiuser scenario. On the other hand, severe decomposition loses object encapsulation and increases reassembly cost.

A balance between data modeling requirements and locking and update characteristics is important for good design. However, in SQL Server, the size of actual stored XML instances is not as critical.

### Limitations of the xml Data Type

Note the following general limitations that apply to the **xml** data type:

* The stored representation of **xml** data type instances cannot exceed 2 GB.
* It cannot be used as a subtype of a **sql\_variant** instance.
* It does not support casting or converting to either **text** or **ntext**. Use **varchar(max)** or **nvarchar(max)** instead.
* It cannot be compared or sorted. This means an **xml** data type cannot be used in a GROUP BY statement.
* It cannot be used as a parameter to any scalar, built-in functions other than ISNULL, COALESCE, and DATALENGTH.
* It cannot be used as a key column in an index. However, it can be included as data in a clustered index or explicitly added to a nonclustered index by using the INCLUDE keyword when the nonclustered index is created.

## FOR XML

A SELECT query returns results as a rowset. You can optionally retrieve formal results of a SQL query as XML by specifying the FOR XML clause in the query.

In a FOR XML clause, you specify one of these modes:

* RAW
* AUTO
* EXPLICIT
* PATH

**Syntax:** FOR { BROWSE | <XML> } ]

<XML> ::=

XML {

{ RAW [ ('ElementName') ] | AUTO }

[

<CommonDirectives>

[ , { XMLDATA | XMLSCHEMA [ ('TargetNameSpaceURI') ]} ]

[ , ELEMENTS [ XSINIL | ABSENT ]

]

| EXPLICIT

[

<CommonDirectives>

[ , XMLDATA ]

]

| PATH [ ('ElementName') ]

[

<CommonDirectives>

[ , ELEMENTS [ XSINIL | ABSENT ] ]

]

}

<CommonDirectives> ::=

[ , BINARY BASE64 ]

[ , TYPE ]

[ , ROOT [ ('RootName') ] ]

Arguments

**RAW[('*ElementName*')]**  
Takes the query result and transforms each row in the result set into an XML element that has a generic identifier, <row />, as the element tag.

**AUTO**  
Returns query results in a simple, nested XML tree. Each table in the FROM clause for which at least one column is listed in the SELECT clause is represented as an XML element.

**EXPLICIT**  
Specifies that the shape of the resulting XML tree is defined explicitly. By using this mode, queries must be written in a particular way so additional information about the nesting you want is specified explicitly.

**PATH**  
Provides a simpler way to mix elements and attributes, and to introduce additional nesting for representing complex properties. You can use FOR XML EXPLICIT mode queries to construct this kind of XML from a rowset, but the PATH mode provides a simpler alternative to the possibly cumbersome EXPLICIT mode queries. PATH mode, together with the ability to write nested FOR XML queries and the TYPE directive to return **xml**type instances, allows you to write queries with less complexity. It provides an alternative to writing most EXPLICIT mode queries.

SELECT ProductId, Name FROM Product FOR XML RAW;

<row ProductId="1" Name="Blanket" />

<row ProductId="2" Name="Mat" />

SELECT ProductId, Name FROM Product FOR XML RAW, TYPE;

<row ProductId="1" Name="Blanket" />

<row ProductId="2" Name="Mat" />

SELECT ProductId, Name FROM Product FOR XML RAW, ELEMENTS;

<row>

<ProductId>1</ProductId>

<Name>Blanket</Name>

</row>

<row>

<ProductId>2</ProductId>

<Name>Mat</Name>

</row>

SELECT ProductID, Name FROM Product FOR XML RAW, ELEMENTS ABSENT

<row>

<ProductID>1</ProductID>

<Name>Blanket</Name>

</row>

<row>

<ProductID>2</ProductID>

<Name>Mat</Name>

</row>

SELECT ProductID, Name FROM Product FOR XML RAW, ELEMENTS XSINIL

<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<ProductID>1</ProductID>

<Name>Blanket</Name>

</row>

<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<ProductID>2</ProductID>

<Name>Mat</Name>

</row>

SELECT ProductID, Name FROM Product FOR XML RAW, ELEMENTS XSINIL ;

**The Name column has the null So it’s displaying xsi:nil**

<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<ProductID>1</ProductID>

<Name>Blanket</Name>

</row>

<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<ProductID>3</ProductID>

<Name xsi:nil="true" />

</row>

# Requesting Schemas as Results with the XMLDATA and XMLSCHEMA Options

SELECT ProductId, Name FROM Product FOR XML RAW,XMLSCHEMA;

<xsd:schema targetNamespace="urn:schemas-microsoft-com:sql:SqlRowSet1" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:sqltypes="http://schemas.microsoft.com/sqlserver/2004/sqltypes" elementFormDefault="qualified">

<xsd:import namespace="http://schemas.microsoft.com/sqlserver/2004/sqltypes" schemaLocation="http://schemas.microsoft.com/sqlserver/2004/sqltypes/sqltypes.xsd" />

<xsd:element name="row">

<xsd:complexType>

<xsd:attribute name="ProductId" type="sqltypes:int" />

<xsd:attribute name="Name">

<xsd:simpleType>

<xsd:restriction base="sqltypes:varchar" sqltypes:localeId="1033" sqltypes:sqlCompareOptions="IgnoreCase IgnoreKanaType IgnoreWidth" sqltypes:sqlSortId="52">

<xsd:maxLength value="100" />

</xsd:restriction>

</xsd:simpleType>

</xsd:attribute>

</xsd:complexType>

</xsd:element>

</xsd:schema>

<row xmlns="urn:schemas-microsoft-com:sql:SqlRowSet1" ProductId="1" Name="Blanket" />

<row xmlns="urn:schemas-microsoft-com:sql:SqlRowSet1" ProductId="2" Name="Mat" />

<row xmlns="urn:schemas-microsoft-com:sql:SqlRowSet1" ProductId="3" />

You can specify the target namespace URI as an optional argument to XMLSCHEMA in FOR XML. This returns the specified target namespace in the schema.

SELECT ProductId, Name FROM Product FOR XML RAW,XMLSCHEMA ('urn:example.com');

<xsd:schema targetNamespace="urn:example.com" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:sqltypes="http://schemas.microsoft.com/sqlserver/2004/sqltypes" elementFormDefault="qualified">

<xsd:import namespace="http://schemas.microsoft.com/sqlserver/2004/sqltypes" schemaLocation="http://schemas.microsoft.com/sqlserver/2004/sqltypes/sqltypes.xsd" />

<xsd:element name="row">

<xsd:complexType>

<xsd:attribute name="ProductId" type="sqltypes:int" />

<xsd:attribute name="Name">

<xsd:simpleType>

<xsd:restriction base="sqltypes:varchar" sqltypes:localeId="1033" sqltypes:sqlCompareOptions="IgnoreCase IgnoreKanaType IgnoreWidth" sqltypes:sqlSortId="52">

<xsd:maxLength value="100" />

</xsd:restriction>

</xsd:simpleType>

</xsd:attribute>

</xsd:complexType>

</xsd:element>

</xsd:schema>

<row xmlns="urn:example.com" ProductId="1" Name="Blanket" />

<row xmlns="urn:example.com" ProductId="2" Name="Mat" />

<row xmlns="urn:example.com" ProductId="3" />

The following query returns the product photo stored in a **varbinary(max)** type column. The BINARY BASE64 option is specified in the query to return the binary data in base64-encoded format.

SELECT ProductPhotoID, ThumbNailPhoto FROM Production.ProductPhoto

WHERE ProductPhotoID=1 FOR XML RAW, BINARY BASE64

<row ProductModelID="1" ThumbNailPhoto="base64 encoded binary data"/>

# Renaming the <row> Element

SELECT ProductId, Name FROM Product FOR XML RAW('Product'), ELEMENTS;

<Product>

<ProductId>1</ProductId>

<Name>Blanket</Name>

</Product>

<Product>

<ProductId>3</ProductId>

</Product>

# Specifying a Root Element for the XML Generated by FOR XML

SELECT ProductId, Name FROM Product FOR XML RAW, ROOT('MYROOT');

<MYROOT>

<row ProductId="1" Name="Blanket" />

<row ProductId="2" Name="Mat" />

<row ProductId="3" />

</MYROOT>

SELECT ProductId, Name FROM Product FOR XML RAW('Product'), ROOT('MYROOT');

<MYROOT>

<Product ProductId="1" Name="Blanket" />

<Product ProductId="2" Name="Mat" />

<Product ProductId="3" />

</MYROOT>

# Querying XMLType Columns

# Use AUTO Mode with FOR XML

 AUTO mode returns query results as nested XML elements. This does not provide much control over the shape of the XML generated from a query result. The AUTO mode queries are useful if you want to generate simple hierarchies. However, [Use EXPLICIT Mode with FOR XML](https://docs.microsoft.com/en-us/sql/relational-databases/xml/use-explicit-mode-with-for-xml?view=sql-server-2017) and [Use PATH Mode with FOR XML](https://docs.microsoft.com/en-us/sql/relational-databases/xml/use-path-mode-with-for-xml?view=sql-server-2017) provide more control and flexibility in deciding the shape of the XML from a query result.

SELECT ProductId, Name ,Color FROM Product ProductDeatils FOR XML AUTO;

<ProductDeatils ProductId="1" Name="Blanket" Color="Black" />

<ProductDeatils ProductId="2" Name="Mat" Color="White" />

<ProductDeatils ProductId="3" />

SELECT ProductId, Name ,Color FROM Product FOR XML AUTO,Elements;

<Product>

<ProductId>1</ProductId>

<Name>Blanket</Name>

<Color>Black</Color>

</Product>

<Product>

<ProductId>2</ProductId>

<Name>Mat</Name>

<Color>White</Color>

</Product>

<Product>

<ProductId>3</ProductId>

</Product>

SELECT Prod.ProductId, Name ,Volume FROM Product Prod JOIN ProductVloume SAL ON Prod.ProductId=SAl.ProductId FOR XML AUTO;

<Prod ProductId="1" Name="Blanket">

<SAL Volume="20" />

</Prod>

<Prod ProductId="2" Name="Mat">

<SAL Volume="30" />

</Prod>

SELECT SAL.ProductId, Prod.Name ,SAL.Volume FROM Product Prod JOIN ProductVloume SAL ON Prod.ProductId=SAl.ProductId FOR XML AUTO,Elements;

<SAL>

<ProductId>1</ProductId>

<Volume>20</Volume>

<Prod>

<Name>Blanket</Name>

</Prod>

</SAL>

<SAL>

<ProductId>2</ProductId>

<Volume>30</Volume>

<Prod>

<Name>Mat</Name>

</Prod>

</SAL>

# Use EXPLICIT Mode with FOR XML

* The first column must provide the tag number, integer type, of the current element, and the column name must be **Tag**. Your query must provide a unique tag number for each element that will be constructed from the rowset.
* The second column must provide a tag number of the parent element, and this column name must be **Parent**. In this way, the Tag and the Parent column provide hierarchy information.

ElementName!TagNumber!AttributeName!Directive

SELECT 1 as Tag, NULL as Parent, NULL [PRODUCT!1!ProductId], NULL [Name!2!Name], NULL [Name!2!Volume] FROM Product Prod JOIN ProductVloume SAL ON Prod.ProductId=SAl.ProductId

UNION ALL

SELECT 2 as Tag, 1 as Parent,Prod.ProductId, Name ,Volume FROM Product Prod JOIN ProductVloume SAL ON Prod.ProductId=SAl.ProductId

FOR XML EXPLICIT;

<PRODUCT />

<PRODUCT>

<Name Name="Blanket" Volume="20" />

<Name Name="Mat" Volume="30" />

</PRODUCT>

# Use PATH Mode with FOR XML

The PATH mode provides a simpler way to mix elements and attributes. PATH mode is also a simpler way to introduce additional nesting for representing complex properties.

# Columns without a Name

SELECT 2+2 FOR XML PATH

<row>4</row>

SELECT ProductId, Name FROM Product FOR XML PATh;

SELECT ProductId, Name FROM Product FOR XML PATh, ELEMENTS;

<row>

<ProductId>1</ProductId>

<Name>Blanket</Name>

</row>

<row>

<ProductId>2</ProductId>

<Name>Mat</Name>

</row>

<row>

<ProductId>3</ProductId>

</row>

SELECT ProductId"@Product", Name"ProductDetails/NAME", Color "ProductDetails/COLOR" FROM Product FOR XML PATh, ELEMENTS;

SELECT ProductId"@Product", Name"ProductDetails/NAME", Color "ProductDetails/COLOR" FROM Product FOR XML PATh;

<row Product="1">

<ProductDetails>

<NAME>Blanket</NAME>

<COLOR>Black</COLOR>

</ProductDetails>

</row>

<row Product="2">

<ProductDetails>

<NAME>Mat</NAME>

<COLOR>White</COLOR>

</ProductDetails>

</row>

<row Product="3" />

SELECT ProductId"@Product", Name"ProductDetails/NAME", Color "ProductDetails/COLOR" FROM Product FOR XML PATh, ELEMENTS XSINIL;

<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" Product="1">

<ProductDetails>

<NAME>Blanket</NAME>

<COLOR>Black</COLOR>

</ProductDetails>

</row>

<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" Product="2">

<ProductDetails>

<NAME>Mat</NAME>

<COLOR>White</COLOR>

</ProductDetails>

</row>

<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" Product="3">

<ProductDetails>

<NAME xsi:nil="true" />

<COLOR xsi:nil="true" />

</ProductDetails>

</row>

SELECT \* FROM Product FOR XML PATh('Productmodel');

<Productmodel>

<ProductId>1</ProductId>

<Name>Blanket</Name>

<Color>Black</Color>

</Productmodel>

<Productmodel>

<ProductId>2</ProductId>

<Name>Mat</Name>

<Color>White</Color>

</Productmodel>

<Productmodel>

<ProductId>3</ProductId>

</Productmodel>