Tamino

Tamino XML Schema User Guide

Version 10.11

November 2021

WEBMETHODS

This document applies to Tamino Version 10.11 and all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: INS-TSL-1011-20211101

Table of Contents

Preface

This document describes:

- The Tamino Schema Language;
- How to model data so that they can be stored and retrieved using Tamino.

This document is intended for:

- Database administrators familiar with methods of data modeling and data management tasks (for example, indexing or mapping). Their task is to create schemas and define them to the Tamino data map.
- Application programmers intending to write programs that address Tamino schemas. They should also be familiar with the principles of schema definition.

The information in this document covers the following topics:

Introduction	Contains basic information about Tamino schemas and the principles of the schema language (TSD) that underlies Tamino.
The Logical Schema	A general description of TSD's logical schema. It explains how you can define elements and attributes and work with various datatypes.

The Schema Header	Discusses the schema header, namely the elements:
	`tsd:schemaInfo`
	`tsd:elementInfo`
	`tsd:attributeInfo`

Tamino-Specific Extensions to the Logical Schema	Describes the extensions to the logical part of TSD that go above and beyond the W3C XML Schema specification.
Tamino-Specific Extensions to the Physical Schema	Describes the extensions to the physical part of TSD.
Schema-Related Attributes in XML Documents	Describes the xsi:type and xsi:nil elements.
Schema Operations	Discusses the operations that can be performed on complete schemas, and constraints for modifying existing schemas.
Tools for the Creation of Tamino Schemas	Introduces tools for working with Tamino schemas.
Appendix 1: Valid Combinations of Restricting Facets and Base Datatypes	Contains information about the facets that are applicable for each datatype.
Appendix 2: Language and Country Codes	Contains information about supported language and country codes.
Appendix 3: Examples	Contains some example schemas.

Appendix 4: Example Schemas for Indexing	Presents some example schemas for indexing.

There is also reference documentation for the TSD elements that are similar to the elements defined in the **W3C XML Schema**, and for TSD's extensions to the **W3C XML Schema** standard:

TSD Logical: Definitions	Reference information for all TSD language elements that belong to the `xs:` namespace.
Tamino Extensions to XML Schema: List of Elements and Attributes	Reference information for all TSD language elements that belong to the `tsd:` namespace.

1 About this Documentation

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
`Monospace font`	Identifies service names and locations in the format *`folder.subfolder.service`*, APIs, Java classes, methods, properties.
`Italic`	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
`Monospace font`	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Software AG Documentation Website

You can find documentation on the Software AG Documentation website at **https://documentation.softwareag.com**.

Software AG Empower Product Support Website

If you do not yet have an account for Empower, send an email to empower@softwareag.com with your name, company, and company email address and request an account.

Once you have an account, you can open Support Incidents online via the eService section of Empower at **https://empower.softwareag.com/**.

You can find product information on the Software AG Empower Product Support website at **https://empower.softwareag.com**.

To submit feature/enhancement requests, get information about product availability, and download products, go to **Products**.

To get information about fixes and to read early warnings, technical papers, and knowledge base articles, go to the **Knowledge Center**.

If you have any questions, you can find a local or toll-free number for your country in our Global Support Contact Directory at **https://empower.softwareag.com/public_directory.aspx** and give us a call.

Software AG Tech Community

You can find documentation and other technical information on the Software AG Tech Community website at **https://techcommunity.softwareag.com**. You can:

- Access product documentation, if you have Tech Community credentials. If you do not, you will need to register and specify "Documentation" as an area of interest.
- Access articles, code samples, demos, and tutorials.
- Use the online discussion forums, moderated by Software AG professionals, to ask questions, discuss best practices, and learn how other customers are using Software AG technology.
- Link to external websites that discuss open standards and web technology.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

2 Introduction

Tamino is an XML-based information server that can manage all types of data. For data storage, Tamino offers its own internal XML store; in addition, Tamino offers the X-Node interface to **Software AG**'s **Adabas**.

A Tamino schema:

- Enables the datatype and structural aspects of an XML instance to be validated as it is stored in Tamino. A schema ensures that every instance stored in a doctype defined in that schema is valid with respect to that schema.
- Defines document types ("doctypes") belonging to a given collection with their respective names and specifies whether they allow the storing of XML or non-XML documents.
- Associates e.g. indexing or collation options with elements and attributes defined in the schema. These options are important for performance and sorting issues.
- Associates mapping information with elements and attributes. This specifies whether they are stored natively in Tamino or (via X-Tension) in an external data store, e.g. Adabas or an SQL database. At query time, these elements and attributes are retrieved from the external database.
- Allows you to specify trigger functions that are invoked when a document is inserted into or deleted from the Tamino data store.

This introduction gives an overview of the issues involved in defining a Tamino schema. It is organized under the following headings:

What is a Schema?

In general, a schema is a description of the structure of XML data. As stated above, a schema is necessary for much of Tamino's functionality, because in many situations Tamino needs to know how or where data is stored in order to perform its tasks optimally.

To express a schema, we need a schema definition language that embraces all the rules necessary for schema definition. For XML, several schema definition languages exist. The most commonly used schema definition languages are:

- Document Type Definition (DTD);
- **XML Schema**.

Although DTDs are still often used to express the structure of an XML document, the newer and more powerful **XML Schema** standard that was defined by the **W3C** is the current schema standard.

The XML Schema Recommendation from the W3C

This section explains the **XML Schema** standard, which is maintained by the **W3C** (World Wide Web Consortium). This standard has evolved to be the most important schema standard beside Document Type Definitions (DTD). Because it is more powerful than DTD, it can be considered as the most important standard for schema definition. Therefore **Software AG** decided to use **XML Schema** as the basis for the Tamino schema definition language (TSD).

W3C Documentation for XML Schema

The official **XML Schema** documentation is divided into three parts:

XML Schema Part 0: Primer
: A non-normative introduction to **XML Schema** explaining the general concepts of XML Schema. It is recommended that you read this document first if you are completely new to XML Schema. It is easier to understand than the other two.

XML Schema Part 1: Structures
: The part of the normative documents that describes structures. It explains the facilities that the **XML Schema** standard offers for describing the structure and constraining the contents of XML 1.0 documents. The schema language considerably extends the capabilities found in DTD.

XML Schema Part 2: Datatypes
: The part of the normative documents that describe the facilities for defining datatypes for XML schemas. It explains the facilities for defining datatypes to be used in XML schemas as well as other XML specifications that exceed the possibilities provided by DTDs by far.

Other Sources of Information on the W3C XML Schema

The following are some non-W3C resources on XML Schema:

- An important source for information and news on XML Schema is Robin Cover's index of XML Schema materials at **http://xml.coverpages.org/schemas.html**
- Roger L. Costello's *XML Schema Tutorial* can be found at **http://www.xfront.com/**

Advanced Tamino Schema Language Based on W3C Standard

The schema language that is implemented in Tamino has been progressively developed, so that it now almost completely covers the **XML Schema** language as defined by the **W3C**, with Tamino-specific extensions. This special subset of **XML Schema** has been customized for Tamino to express (nearly) all DTD capabilities in **XML Schema** syntax plus the most important features for storing XML data.

In Tamino, you define a schema by the following two-step procedure:

To define a schema

1. Specify the structural information of the schema using the **XML Schema** language. This can, for example, be done with the Tamino Schema Editor. If you already have a DTD describing your doctype, the Tamino Schema Editor can convert it into a TSD Schema. Otherwise, you must write the **XML Schema** from scratch.

   This is done using the logical part of TSD.

   The appropriate elements of the logical part of the TSD are described in the sections ***The Logical Schema***, ***Tamino-Specific Extensions to the Logical Schema*** and *TSD Logical: Definitions*.
2. Specify physical storage information (for example, for mapping and indexing) using the Tamino-specific extensions to **XML Schema**.

   This is done using the physical part of TSD.

   The appropriate elements of the physical part of the TSD are described in the sections ***Tamino-Specific Extensions to the Physical Schema*** and *Tamino Extensions to XML Schema: List of Elements and Attributes*.

For more information, see the section ***Creating a Schema***.

For an overview of the Tamino schema elements and attributes, see the Tamino XML Schema Reference Guide.

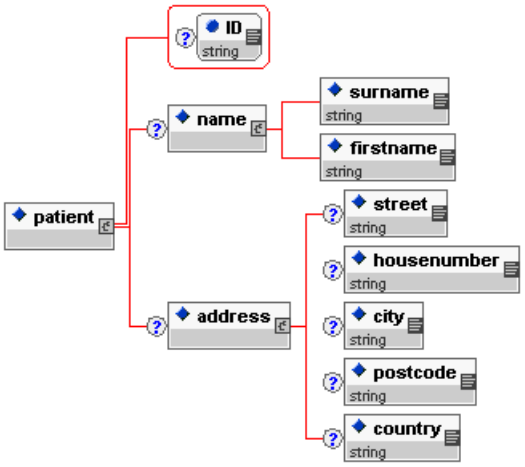
TSD implements all features of the W3C **XML Schema** with the following exceptions:

- `<xs:redefine>` is not implemented.
- The so-called “chameleon include” is not supported. A chameleon include occurs when a schema without a target namespace is included, via `<xs:include>`, by a schema with a non-absent target namespace, thus inheriting the including schema's target namespace.
- The instance attributes `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` are ignored by Tamino.

See also the section *Features of the W3C XML Schema that are Not Supported by Tamino*.

Example

The following figure illustrates the schema of a patient record (doctype “patient”) in a hospital database:



This tree structure can be represented by the following XML schema, which contains only structural information:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <!-- Schema for patient data.
       This could be a fragment of a larger DTD modeling hospital data -->
  <xs:element xs:name='patient'>
    <xs:complexType>
      <xs:sequence>
        <xs:element xs:ref='name' xs:minOccurs='0' />
        <xs:element xs:ref='address' xs:minOccurs='0' />
      </xs:sequence>
      <xs:attribute xs:name='ID' xs:type='xs:string' xs:use='optional' />
    </xs:complexType>
  </xs:element>
  <xs:element xs:name='name'>
    <xs:complexType>
      <xs:sequence>
        <xs:element xs:ref='surname' />
        <xs:element xs:ref='firstname' />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element xs:name='surname' xs:type='xs:string' />
  <xs:element xs:name='firstname' xs:type='xs:string' />
  <xs:element xs:name='address'>
    <xs:complexType>
```

```
      <xs:sequence>
        <xs:element xs:ref='street' xs:minOccurs='0' />
        <xs:element xs:ref='housenumber' xs:minOccurs='0' />
        <xs:element xs:ref='city' xs:minOccurs='0' />
        <xs:element xs:ref='postcode' xs:minOccurs='0' />
        <xs:element xs:ref='country' xs:minOccurs='0' />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element xs:name='street' xs:type='xs:string' />
  <xs:element xs:name='housenumber' xs:type='xs:string' />
  <xs:element xs:name='city' xs:type='xs:string' />
  <xs:element xs:name='postcode' xs:type='xs:string' />
  <xs:element xs:name='country' xs:type='xs:string' />
</xs:schema>
```

Fundamentals of Tamino Schema Definition

Conventions Used in this Document

The following prefixes are used for namespaces that occur frequently throughout this documentation:

- tsd: *http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition*
- xs: *http://www.w3.org/2001/XMLSchema*
- xsi: *http://www.w3.org/2001/XMLSchema-instance*

The xs:annotation/xs:appinfo Mechanism for Adding Tamino-Specific Extensions

The XML Schema Recommendation allows any schema document to be extended by arbitrary content in `xs:annotation` and `xs:appinfo` elements that are children of any other element defined by **XML Schema**.

TSD takes advantage of this feature by adding elements `tsd:schemaInfo`, `tsd:elementInfo` and `tsd:attributeInfo` as children of the `xs:schema`, `xs:element` and `xs:attribute` elements respectively. These extensions belong to the namespace *http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition*. Only one `tsd:schemaInfo`, `tsd:elementInfo` or `tsd:attributeInfo` element is allowed for each schema, element or attribute definition respectively.

Example Schema

The following Tamino schema code snippet illustrates how this mechanism can be used to incorporate schema-related information in the `xs:schema` element:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema"
           xmlns:tsd = ↩
"http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
  <xs:annotation>
    <xs:appinfo>
      <tsd:schemaInfo name = "patient">
        <tsd:collection name = "hospital"/>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
  .
  .
  .
</xs:schema>
```

Example Element

Tamino-related extensions attached to a single element are specified as illustrated in the following example:

```
<xs:element name="surname">
  <xs:annotation>
    <xs:appinfo>
      <tsd:elementInfo>
        .
        .
        .
      </tsd:elementInfo>
    </xs:appinfo>
  </xs:annotation>
  .
  .
  .
</xs:element>
```

Example Attribute

Tamino-related extensions attached to a single attribute are specified as illustrated in the following example:

```
<xs:attribute name="value">
  <xs:annotation>
    <xs:appinfo>
      <tsd:attributeInfo>
        .
        .
        .
      </tsd:attributeInfo>
    </xs:appinfo>
  </xs:annotation>
  .
  .
  .
</xs:attribute>
```

For more information concerning this topic, please refer to the following:

- The section *Tamino Annotations in XML Schema* of the *Advanced Concepts* manual;
- The description of the `xs:annotation` element;
- The description of the `xs:appinfo` element.

Meta Schema

The Tamino Schema Definition Language is defined by a meta schema, or schema of schemas, which is based on a subset of the W3C **XML Schema** standard. The TSD meta schema, i.e. the schema of schemas, can itself be expressed in terms of **XML Schema**.

Limitations on the Lengths of Names

The following applies:

- The names of collections and doctypes are of type `xs:NMTOKEN`.
- The names of elements and attributes are of type `xs:NCName`.

The following restriction on the lengths of names apply:

- The name of a collection, doctype, element or attribute can have up to 240 bytes when encoded as UTF-8. (In practice, this corresponds to at least 80 characters.)

Creating a Schema

Defining a schema for storing data in Tamino means choosing the kind of storage (for example, native or non-XML storage or mapping to Adabas, Server Extension) and determining the kind of indexing to be performed on the data. Additionally, triggers and unique constraints can be defined. Normally, you should choose native storage to take advantage of the full power of Tamino.

However, there is more than one way to store an XML object in Tamino's XML store. You can:

- Store a well-formed XML object without defining a schema. Tamino stores it either in a special area, namely the collection `ino:etc`, or in a user-defined collection that has been enabled especially for schemaless data storage. Tamino applies default indexing that allows full text retrieval.

  Note: For more information, see the section *Instances Without a Defined Schema* and especially the subsection *Using User-Defined Collections for Storing Instances Without a Defined Schema*.

- Directly edit the Tamino schema of which the XML object is an instance using the Tamino Schema Editor.
- Directly edit the XML schema with XML schema tools, as long as you keep to the XML schema subset defined by TSD.
- Read the DTD of which the XML object is an instance into the Tamino Schema Editor or any another schema conversion tool (third party software) and generate a schema based on the DTD. You can define indices in order to optimize certain types of retrieval operations. When you store an instance of the DTD in Tamino, indexing as defined in the schema is applied.
- Write a schema using a text editor. This route is error-prone because of the relative complexity of the schema syntax. The instance must conform to the schema, otherwise it cannot be stored.

See also the section *Tools for the Creation of Tamino Schemas*.

Further Documentation

The documentation of the TSD language comprises the section you are currently reading and the Tamino XML Schema Reference Guide, which describes each element of the language.

Related documentation

- *Tamino Schema Editor*;
- *Advanced Concepts*;
- *XML Namespaces in Tamino*.

3 The Logical Schema

The following topics are discussed in this chapter:

Schema

The root element of each **XML Schema**-conformant schema document is the `xs:schema` element. It may have the optional `targetNamespace` attribute, which specifies the namespace to which all definitions in the current schema document belong.

```
<xs:schema targetNamespace="http://my-company.com/"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

If a schema has a `targetNamespace`, all globally defined objects (elements, attributes, type definitions etc.) that occur as direct child elements of `xs:schema` have qualified names that belong to that namespace. Locally declared elements or attributes only belong to the `targetNamespace` if their `form` attribute, which defaults to the value of the `elementFormDefault` or `attributeFormDefault` attribute of `xs:schema`, has a value of "qualified".

Elements and Attributes

Elements and attributes are described by:

- **Elements** `xs:element` **and** `xs:attribute`
  If these elements are child elements of the `xs:schema` root element, they are called *global* elements or attributes, respectively. Otherwise, we speak of *local* elements or attributes.
- **The** `name` **attribute**
  The value of the `name` attribute is an unqualified (i.e. local) name (without namespace prefix) of the element or attribute being defined, or a `ref` attribute containing a qualified reference to a global element or attribute (i.e. containing the namespace prefix, if relevant).
- **Cardinality**
  The cardinality of the `xs:element` can be specified using `minOccurs` (default value: 1) and `maxOccurs` (default value: 1; the value may also be "unbounded") attributes. Similarly, the cardinality of `xs:attribute` is specified by the `use` attribute: `use="optional"` (default) or `use="required"`.
- **Optional** `type`**attribute**
  Only one of the following variants can be used.
  - The `type` attribute is a qualified reference to a globally defined type. This may be one of the following set of more than 40 simple types predefined by **XML Schema** a user-defined named simple type or user-defined complex type.

Example:

```
<xs:element name="fee" type="xs:decimal" />
```

Instead of using the `type` attribute, one of the child elements `xs:simpleType` or `xs:complexType` may be used to define the type of the logical node.

- `xs:simpleType` allows you to add constraints via facets specified as child elements of a nested `xs:restriction` element.

Example:

This example shows an element of predefined type, `xs:string`, with the `xs:maxLength` facet restricting the maximum length of the string; it has no attributes.

```
<xs:element name="surname">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:maxLength value="20"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Alternatively, a new simple type can be constructed using `xs:list` or `xs:union`.

```
<xs:simpleType name="listOfInt">
  <xs:list itemType="xs:int" />
</xs:simpleType>
```

validates any list of whitespace-separated values of type `xs:int`:

```
<xs:simpleType name="typeOfMaxOccurs">
  <xs:union memberType="xs:nonNegativeInteger">
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="unbounded" />
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
```

shows a possible type definition for the `maxOccurs` attribute of `xs:element`; it allows for a non-negative integer or the token "unbounded".

- Complex type definitions are described **below**.

Note: A complex type can only be specified for element declarations; it is not allowed for attribute declarations. Only an element declaration may specify or reference a complex type definition.

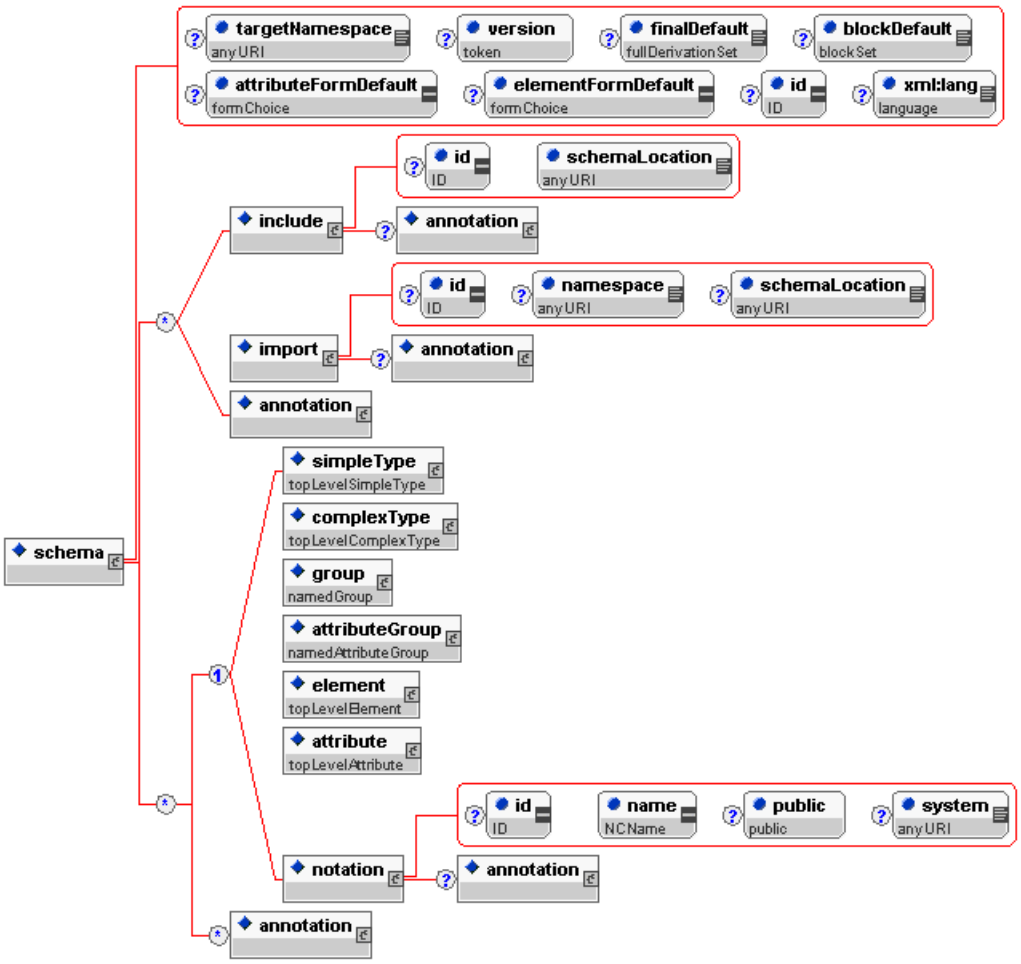
Overview: The Logical Part of the Meta Schema

The logical schema specifies the structural information of the schema.

The Tamino Schema Language is defined by a meta schema (also known as a schema of schemas), based on a subset of the W3C **XML Schema** standard. It is described in the section *TSD Logical: Definitions* of the *Tamino XML Schema Reference Guide*.

The structure of the logical schema is shown in the graphics below. The root element is the `xs:schema` element.

Note: The namespace prefix `xs:` is omitted in all of these graphics.



Logical part of TSD: Expansion of root element xs:schema

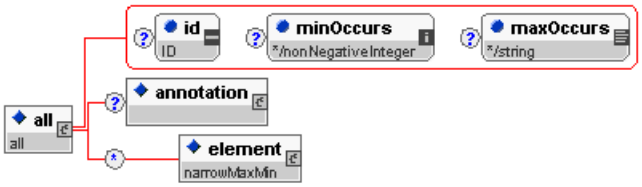
The top-level element `xs:schema` as defined in this graphic is the container for all the information pertaining to a Tamino schema. For further information about the elements and attributes as they are defined in the W3C standard, see **XML Schema Elements**. This section describes Tamino's implementation of the standard.

Particles

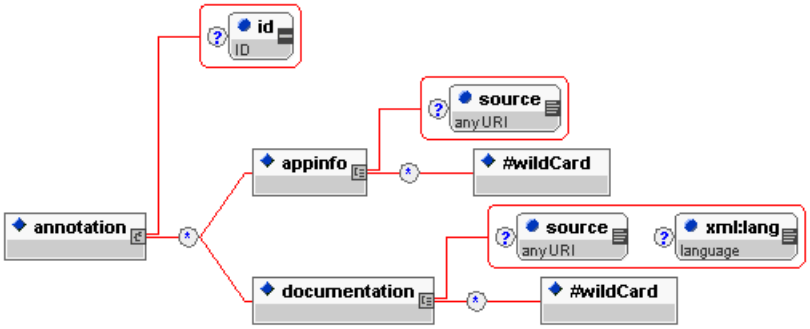
The following particle definitions appear below:

1. `xs:all`
2. `xs:annotation`
3. `xs:any`
4. `xs:choice`
5. `xs:element`
6. `xs:field`

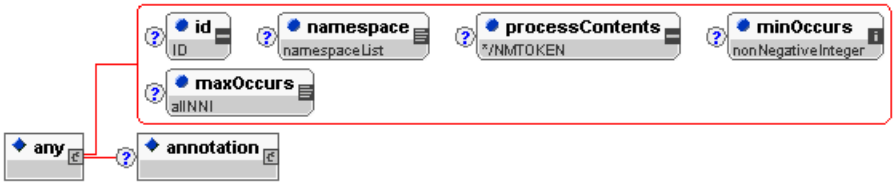
7. `xs:group`
8. `xs:selector`
9. `xs:sequence`



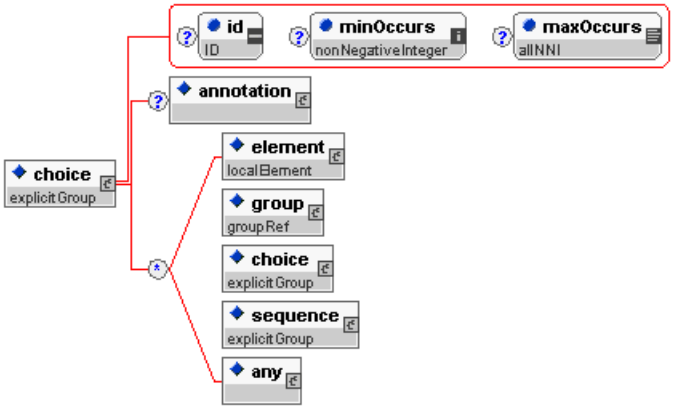
Logical part of TSD: Expansion of xs:all



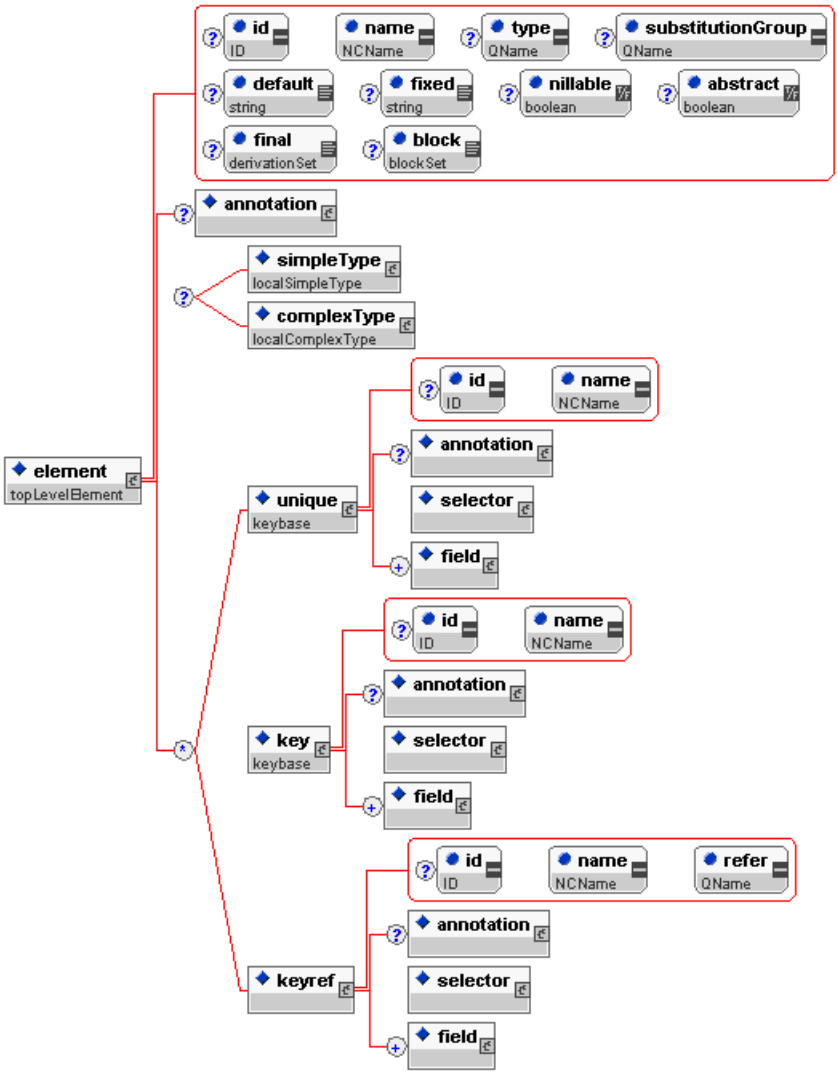
Logical part of TSD: Expansion of xs:annotation



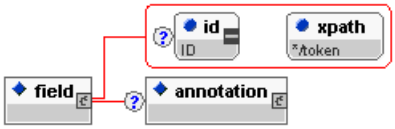
Logical part of TSD: Expansion of xs:any



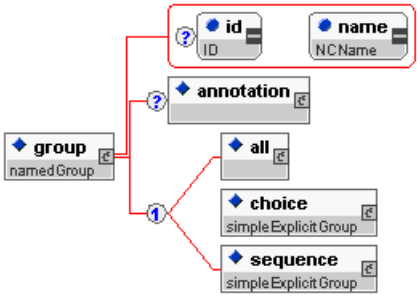
Logical part of TSD: Expansion of xs:choice



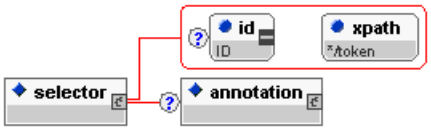
Logical part of TSD: Expansion of xs:element



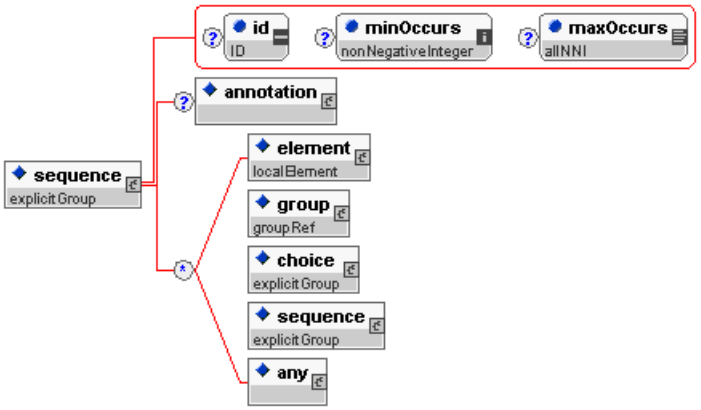
Logical part of TSD: Expansion of xs:field



Logical part of TSD: Expansion of xs:group



Logical part of TSD: Expansion of xs:selector



Logical part of TSD: Expansion of xs:sequence

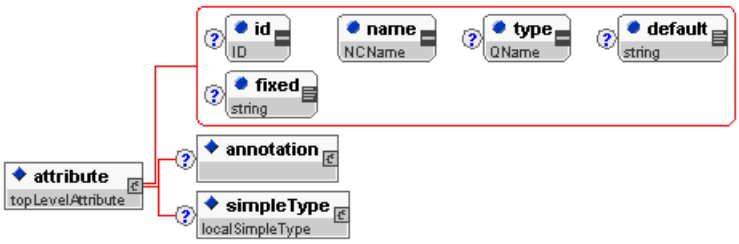
Attribute-Related Elements

The following attribute element definitions appear below:

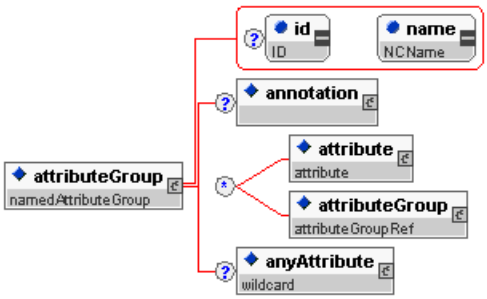
1. `xs:anyAttribute`
2. `xs:attribute`

3. `xs:attributeGroup`

Logical part of TSD: Expansion of xs:anyAttribute



Logical part of TSD: Expansion of xs:attribute

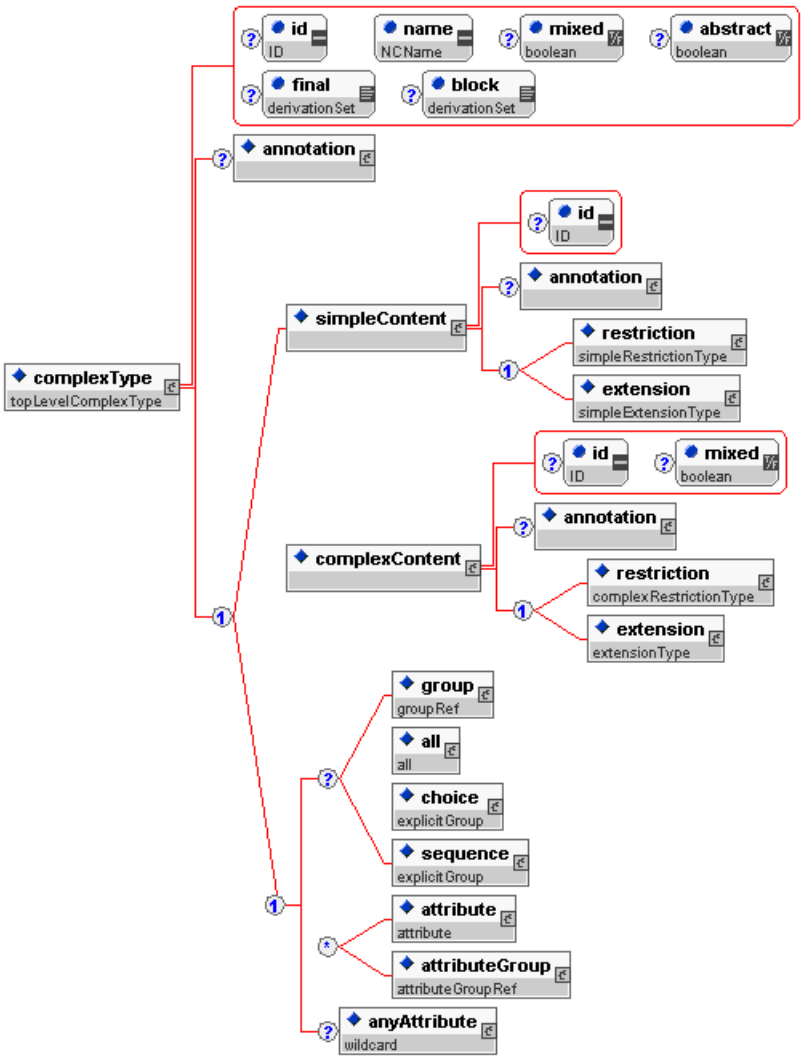


Logical part of TSD: Expansion of xs:attributeGroup

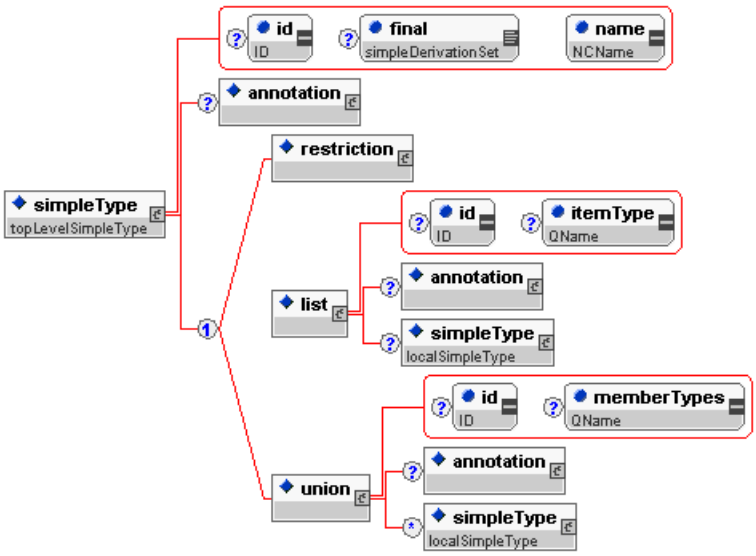
Type-Related Elements

The following type element definitions appear below:

1. `xs:complexType`
2. `xs:simpleType`



Logical part of TSD: Expansion of xs:complexType

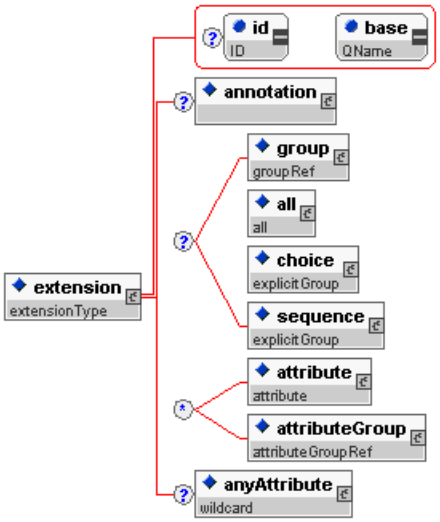


Logical part of TSD: Expansion of xs:simpleType

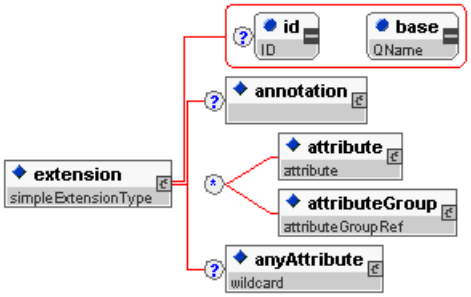
Type Derivation Elements

The following type derivation element definitions appear below:

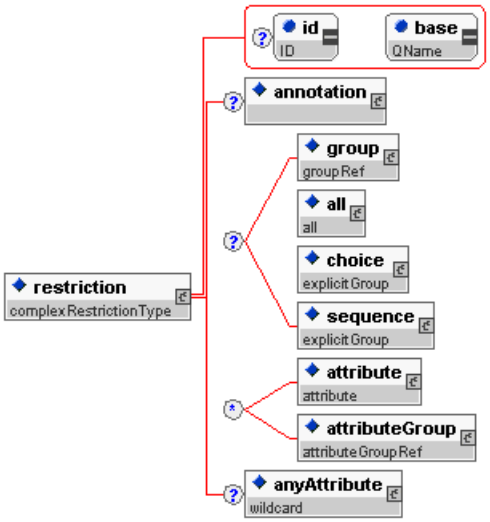
1. `xs:extension` as a child element of `xs:complexContent`
2. `xs:extension` as a child element of `xs:simpleContent`
3. `xs:restriction` as a child element of `xs:complexContent`
4. `xs:restriction` as a child element of `xs:simpleContent`



Logical part of TSD: Expansion of xs:complexContent/xs:extension



Logical part of TSD: Expansion of xs:simpleContent/xs:extension



Logical part of TSD: Expansion of xs:complexContent/xs:restriction

Logical part of TSD: Expansion of xs:simpleContent/xs:restriction

Note: `xs:restriction` as a child element of `xs:simpleType` has the same content model as `xs:restriction` as a child element of `xs:simpleContent`, except that it does not allow `xs:simpleType`, `xs:attribute`, `xs:attributeGroup` and `xs:anyAttribute` as child elements.

Constraints on Element Definition

In general, there are two kinds of elements in **XML Schema**: elements of **simple type** and elements of **complex type**. Elements of complex type contain other elements or attributes, whereas elements of simple type contain only character data but neither child elements nor attributes. `xs:element` enables you to define elements of both simple type and complex type by using its child elements `xs:simpleType` and `xs:complexType`, or by referencing a user-defined or predefined named type. Lists of the predefined types offered by the **XML Schema** standard can be found at *http://www.w3.org/TR/xmlschema-0/#CreatDt* and *http://www.w3.org/TR/xmlschema-2/#built-in-datatypes*. All the predefined simple types are also available in Tamino. Additionally, in **XML Schema** there are mechanisms for deriving new types from existing types. These are:

- extension;
- restriction;
- list;
- union.

Type derivation by restriction can be used:

- to introduce constraining facets, in the case of simple types or complex types with simple content;
- to restrict attribute occurrences for complex types with simple content or complex content;
- to restrict the content model, in the case of complex types with complex content;
- to apply constraining facets to the original datatype (base datatype).

The restriction mechanism is provided by the `xs:restriction` element.

Complex type definitions offer the possibility of defining elements with attributes, and elements with child elements. The **XML Schema** standard offers a wealth of possibilities for defining elements and attributes. Most but not all of them are available in the Tamino schema definition language.

In general, TSD (like **XML Schema**) offers both named and anonymous complex type definitions, i.e. a complex type definition may or may not have a name attribute by which it can subsequently be referenced.

The complex type definition of **XML Schema** is described in detail at *http://www.w3.org/TR/xmlschema-0/#DefnDeclars*

An attribute can be defined using the extension mechanism. For defining extensions, use the `xs:extension` child element of the `xs:simpleContent`, which is, in turn, a child element of the `xs:complexType` element. Here is an example:

```
<xs:complexType>
  <xs:simpleContent>
    <xs:extension base = "xs:normalizedString">
      <xs:attribute name = "duration"
                    type = "xs:unsignedShort"
                    use = "required"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Here, a new complex type is created by extending the existing type `xs:normalizedString` with an attribute whose name is `duration` that must be specified (`use = required`) and is of type `xs:unsignedShort`. The `xs:simpleContent` element indicates that the element to be defined has no child elements, i.e. it contains only character data and attributes.

You can create a complex type containing elements by using, for example, the `xs:sequence` element, which allows you to define a sequence of elements comprising the content model. Also you can specify alternatively appearing elements with the `xs:choice` element. If you have defined elements using the `xs:sequence`, `xs:choice` or `xs:all` elements and want to define additional attributes, this cannot be done as described above using the `xs:simpleContent` element; however, it can be achieved by the `xs:attribute` element.

Tamino also supports groups and attribute groups as defined by **XML Schema**. For more information, see the descriptions of the elements `xs:group` and `xs:attributeGroup`.

Constraints on Attribute Definition

Attributes can be declared locally or globally. They can only be of simple types. Therefore, the constraints on attributes (which correspond to the constraints on elements described above) only apply to the attribute `type` and the element `xs:simpleType`.

- **Attribute with Simple Type**

  Attribute: `type`.

- **Attribute with Simple Type and Facets**

  Element `xs:simpleType` with element `xs:restriction` containing **facets**.

- **Reference to Attributes**

  Attribute: `ref`.

For more information, see the section *xs:attribute element* of the *Tamino XML Reference Guide*.

XML Datatypes

This section describes Tamino's facilities for specifying datatypes for XML data. It is subdivided into the following parts:

Datatype and Facet Support in TSD Compared to XML Schema 1.0

Tamino offers the same possibilities for defining new datatypes as **XML Schema** does; of these, the simple types are described in the **W3C** standards document ***XML Schema Part 2: Datatypes***. All datatypes described in that document are implemented in TSD.

For readers not familiar with **XML Schema**, the basics of the mechanisms used for datatype definition in Tamino and **XML Schema** are briefly summarized here.

According to the definition in the **XML Schema** specification, a **datatype** is a set (more precisely, a 3-tuple) comprising the following items:

The value space
: The value space is the set of values that is allowed for a given datatype. A value space has some properties; for example, it can be ordered.

The lexical space
: The lexical space is the set of valid literals for a datatype. It may be possible to have more than one representation for one and the same member of a specific value space. These different representations are different members in the lexical space that represent the same element in the value space. For example, "1000" and "1.000E3" are two different literals and therefore two different members of the lexical space that both represent the same member in the value space of the datatype `float`.

A set containing one or more facets
: A facet is a property of a value space that can be used to characterize that value space. It typically represents a single aspect of the value space (i.e., a single dimension in a multi-dimensional representation of the value space). A facet can be fundamental (semantically character-

izing the value space) or non-fundamental (defining constraints to the value space, therefore also called a constraining facet).

Built-In Datatypes

These datatypes are predefined in Tamino (an in the W3C **XML Schema**), so you can use them without having to declare or define them. They can be divided into two groups:

- ***Primitive types in Tamino***
- ***Derived built-in datatypes***

Primitive Types in Tamino

Primitive datatypes are predefined by the **XML Schema** standard. The following primitive types are available in Tamino:

Datatype	Description	Lexical Representation
`xs:string`	Character string of unlimited length	A short string
`xs:boolean`	Boolean value.	"true", "false", "1", "0"
`xs:decimal`	Decimal number. A precision of at least 18 digits is supported.	"-1.23", "125.64", "0.0", "+500000.00", "170"
`xs:float`	Single-precision 32-bit floating point type according to the ***IEEE 754-1985 Standard for Binary Floating-Point Arithmetic***. This type includes the special values positive and negative zero, positive and negative infinity, and not-a-number.	"-1E3", "172.363E14", "18.73e-5", "45", "INF", "-INF", "0", "-0", "NaN"
`xs:double`	Double-precision 64-bit floating point type according to the ***IEEE 754-1985 Standard for Binary Floating-Point Arithmetic***.	"-1E4", "547.433E12", "36.78e-2", "12", "INF", "-INF", "0", "-0", "NaN"
`xs:duration`	This datatype specifies a period of time: The value space is a six-dimensional space, where the coordinates designate the Gregorian year, month, day, hour, minute and second. **Note:** This datatype cannot indexed.	The lexical representation follows the format "PnYnMnDTnHnMnS". An optional fractional part for seconds is allowed. Negative durations are also allowed.
`xs:time`	A specific time of day as defined in §5.3 of the ***ISO 8601 standard*** on date and time formats. Also see note below.	The lexical format is `hh:mm:ss` **Note:** An optional fractional part for seconds is permitted. A time zone can be specified, if necessary: "Z" for UTC time, or a signed time difference in the format `hh:mm`

Datatype	Description	Lexical Representation
		Examples: "05:20:23.2" "13:20:00-05:00"
xs:date	A Gregorian calendar date according to §5.2.1 of the *ISO 8601 standard* on date and time formats. Also see note below.	The lexical format is CCYY-MM-DD. To accommodate values outside the range 1-9999, additional digits and a negative sign can be added to the left. (The year 0000 is prohibited.) **Example:** "1999-05-31"
xs:dateTime	A specific instant of time (a combination of date and time) as defined in §5.4 of the *ISO 8601 standard* on date and time formats. Also see note below.	The lexical format is CCYY-MM-DDThh:mm:ssZ, where "T" is the delimiter character between date and time and "Z" denotes an optional time zone. **Examples:** "1999-05-31T13:20:00-05:00" "2001-12-01T05:20:23.2"
xs:gYearMonth	This datatype represents a specific Gregorian month in a specific Gregorian year.	The lexical format is CCYY-MMZ, where "Z" denotes an optional time zone. **Example:** "2001-05"
xs:gYear	This datatype represents a Gregorian year.	The lexical format is CCYYZ, where "Z" denotes an optional time zone. **Example:** "1994"
xs:gMonthDay	This datatype specifies a Gregorian date.	The lexical format is --MM-DDZ, where "Z" denotes an optional time zone. **Example:** "--04-01"
xs:gMonth	This datatype denotes a Gregorian month that recurs every year.	The lexical format is --MM--Z, where "Z" denotes an optional time zone. **Example:** "--07" ("--07--" is accepted for backward compatibility)

Datatype	Description	Lexical Representation
xs:gDay	This datatype denotes a Gregorian day that recurs every month.	The lexical format is ---DDZ, where "Z" denotes an optional time zone. **Example:** "---13"
xs:hexBinary	Hexadecimal-encoded arbitrary binary data.	**Examples:** "9a7f", "FFFF3", "0100"
xs:base64Binary	Base64-encoded arbitrary binary data. The entire binary stream is encoded using the Base64 Content-Transfer-Encoding defined in Section 6.8 of *RFC 2045*.	
xs:anyURI	A reference to a Uniform Resource Identifier (URI).	
xs:QName	An XML qualified name, consisting of a namespace name and a local part.	
xs:NOTATION	Represents the NOTATION attribute type from XML attributes.	This is an abstract datatype, i.e. the user must derive an own datatype from it.

Derived Built-In Datatypes

In addition to these primitive datatypes, it is also possible to derive datatypes in TSD from other datatypes, which in turn may be either primitive types or derived types, by two different mechanisms:

- **Restriction**
  This means that the value space of the original datatype is constrained in some respect by a constraining facet. For example, the datatype nonNegativeInteger is derived from the datatype integer by constraining its value space to non-negative values; the constraint consists of the exclusion of negative values.
- **List**
  This offers the possibility of creating a new datatype by combining elements of existing datatypes: instead of a single element of the datatype in question, a sequence of values of the same datatype is allowed. For example, the datatype xs:NMTOKENS is derived from the datatype xs:NMTOKEN by constructing a list of NMTOKEN values.

The W3C **XML Schema** standard contains further mechanisms for deriving datatypes from a given datatype.

The following derived datatypes are supported in Tamino:

Datatype	Derived From	Description
xs:normalizedString	String	A string after whitespace normalization.
xs:token	xs:normalizedString	Does not contain the line feed ("#xA") or tab ("#x9") characters, d not have leading or trailing spaces ("#x20") and does not have multiple consecutive internal spaces.
xs:NMTOKEN	xs:token	Represents the **NMTOKEN attribute type** (DTD) that is descri in **http://www.w3.org/TR/2004/REC-xml-20040204/#NT-Nmtok**
xs:NMTOKENS	xs:NMTOKEN	Represents the **NMTOKENS attribute type** (DTD) that is descri in **http://www.w3.org/TR/2004/REC-xml-20040204/#NT-Nmtok**
xs:Name	xs:token	Represents an **XML Name** as described in **http://www.w3.org/TR/2004/REC-xml-20040204/#NT-Name**.
xs:NCName	xs:Name	Represents an **XML "non-colonized" Name** as described in **http://www.w3.org/TR/xmlschema-2/#NCName**.
xs:ID	xs:NCName	Represents the **ID attribute type** as described in **http://www.w3.org/TR/xmlschema-2/#ID**.
xs:IDREF	xs:NCName	Represents the **IDREF attribute type** as described in **http://www.w3.org/TR/xmlschema-2/#IDREF**.
xs:IDREFS	xs:IDREF	Represents the **IDREFS attribute type** as described in **http://www.w3.org/TR/xmlschema-2/#IDREFS**.
xs:ENTITY	xs:NCName	Represents the **ENTITY attribute type** as described in **http://www.w3.org/TR/xmlschema-2/#ENTITY**.
xs:ENTITIES	xs:ENTITY	Represents the **ENTITIES attribute type** as described in **http://www.w3.org/TR/xmlschema-2/#ENTITIES**.
xs:language	xs:token	Represents formal language identifiers, as defined by RFCs 30 4646 and 4647 or their successor(s). The value space and lexica space are the set of all strings that conform to the pattern `[a-zA-Z]{1,8}(-[a-zA-Z0-9]{1,8})*`.
xs:integer	xs:decimal	The standard mathematical integer datatype. Derived from datat decimal by setting the facet "fractionDigits" to 0.
xs:nonPositiveInteger	xs:integer	An integer less than or equal to zero.
xs:negativeInteger	xs:nonPositiveInteger	An integer less than zero.
xs:long	xs:integer	An integer in the range -9223372036854775808 (-2^{63}) to 9223372036854775807 (2^{63}-1).
xs:int	xs:long	An integer in the range -2147483648 (-2^{31}) to 2147483647 (2^{31}-1)

Datatype	Derived From	Description	SQL Equ
xs:short	xs:int	An integer in the range -32768 (-2^{15}) to 32767 (2^{15}-1).	SMA
xs:byte	xs:short	An integer in the range -128 (-2^{7}) to 127 (2^{7}-1).	TIN
xs:nonNegativeInteger	xs:integer	An integer greater than or equal to zero (0 to 2^{64}-1).	no equ
xs:unsignedLong	xs:nonNegative**Integer**	An integer in the range 0 to 2^{64}-1.	no equ
xs:unsignedInt	xs:unsignedLong	An integer in the range 0 to 4294967295 (2^{32}-1).	no equ
xs:unsignedShort	xs:unsignedInt	An integer in the range 0 to 65535 (2^{16}-1).	no equ
xs:unsignedByte	xs:unsignedShort	An integer in the range 0 to 255 (2^{8}-1).	TIN
xs:positiveInteger	xs:nonNegative**Integer**	An integer greater than zero.	no equ

User-Defined Datatypes

You can create two kinds of user-defined datatypes in Tamino (or **XML Schema**):

- **Simple Datatypes**
  Simple datatypes are datatypes that users of Tamino or **XML Schema** can define by themselves. An element that is defined using a simple datatype can have neither child elements nor attributes. In Tamino, a simple datatype can be defined with the `xs:simpleType` element.

  In Tamino's Schema Definition Language TSD, a new datatype can be specified as shown in the following example using an XML Schema simple type definition:

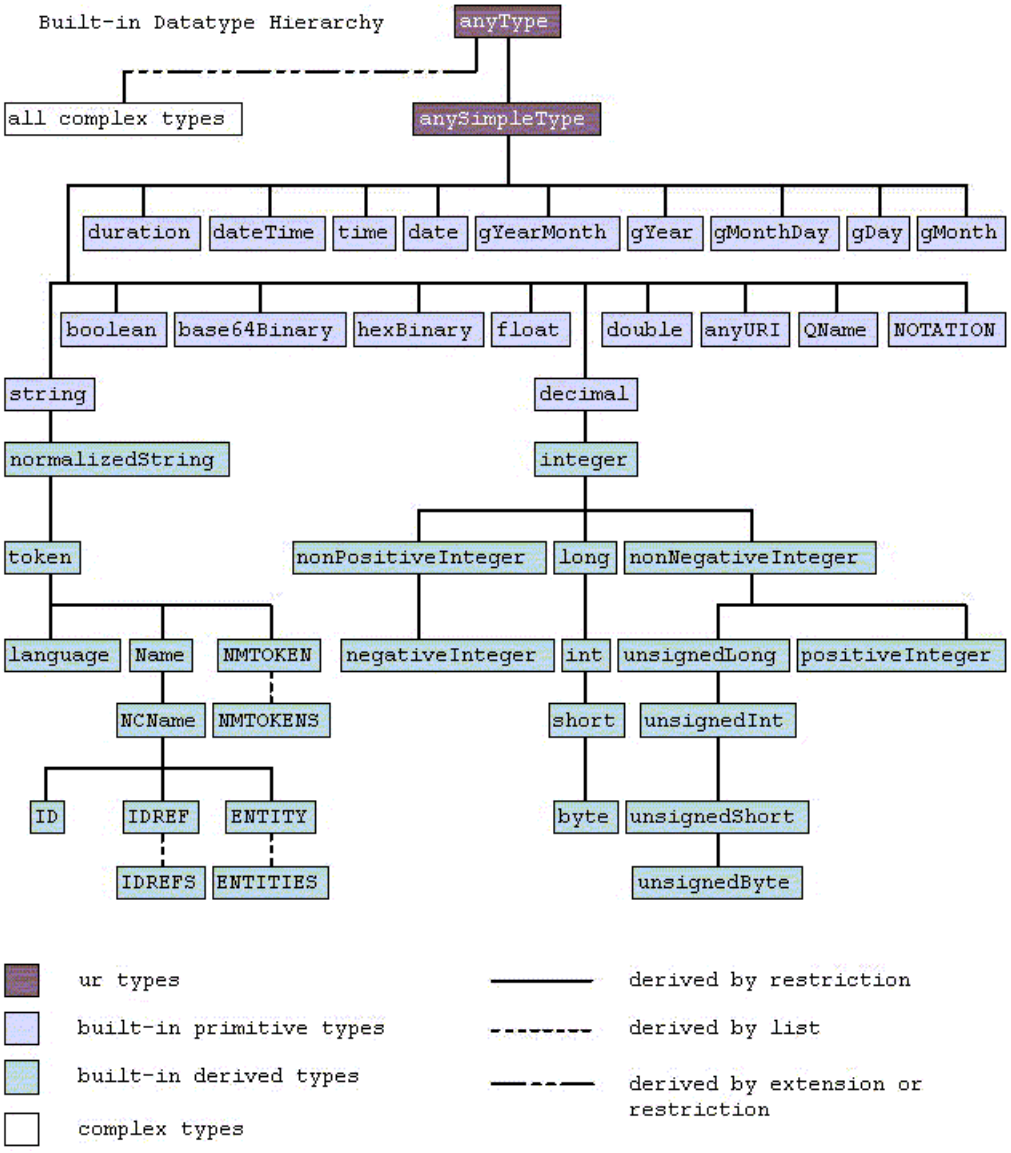
```
<xs:element name = "---">
  <xs:simpleType>
    <xs:restriction base="xs:decimal">
      <xs:totalDigits value= "15"/>
      <xs:fractionDigits value = "5"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

  This defines a decimal datatype with the precision set to 15 digits and the number of fraction digits set to 5.

- **Complex Datatypes**
  In contrast to the simple datatype definitions, complex type definitions can be used to define elements that contain child elements and/or attributes. This very powerful technique is realized in Tamino with the `xs:complexType` element.

Hierarchy of Datatypes

The hierarchy of datatypes allowed in TSD is depicted in the following graphic:



Built-in Datatype Hierarchy

Ranges of Numeric Types and Related Issues

The value space of the datatype `integer` is from -9223372036854775808 (-2^63, approx. -9E18) to +9223372036854775807 (2^63-1, approx. 9E18). Numbers outside this range lead to overflow errors. In queries, the value -9223372036854775808 must be coded as (-9223372036854775807 -1). This type is sometimes called “signed integer”.

The value space of the datatype `unsigned integer` is from 0 to 18446744073709551615 (2^64, approx. 1.8E19). Numbers outside of this range will lead to overflow errors.

The value space of the datatype `decimal` is from -999999999999999999 (-1E18-1) to +999999999999999999, or more precisely: from -999999999999999999 to -0.000000000000000001, 0, and from 0.000000000000000001 to +999999999999999999. Accuracy is limited to 18 significant digits; for example, 123456789.987654321 (18 significant digits) can be represented exactly, but 123456789.9876543215 (19 significant digits) is rounded to 123456789.987654322. Numbers between -0.000000000000000005 and 0.000000000000000005 are rounded to zero. Numbers greater than or equal to 999999999999999999.5 or less than or equal to -999999999999999999.5 lead to overflow.

The value spaces of the datatypes `float` and `double` and their binary representations are as specified in *IEEE* 754. All comparisons and arithmetic operations with numeric data are carried out in the internal representation. Consequently, the limitations of binary representation as described in IEEE 754: ***IEEE Standard for Binary Floating-Point Arithmetic*** apply.

If you want to use the datatypes `float` or `double`, you should understand these numeric formats, e.g. by reading an introductory text on numerical mathematics, in order to know exactly the difficulties and limitations.

If you have to calculate financial results, use the type `decimal`. Neither `float` nor `double` is suitable.

The approximate ranges of `float` and `double` in decimal notation are as follows:

The range of `float` is approximately from -3.402823466E+38 to -1.175494351E-38, 0, and from 1.175494351E-38 to 3.402823466E+38; also the special values -INF, INF, NaN.

The range of `double` is approximately from -1.7976931348623158E+308 to -2.2250738585072014E-308, 0, and from 2.2250738585072014E-308 to 1.7976931348623158E+308; also the special values -INF, INF, NaN.

Numbers outside these ranges lead to overflow or to the results -INF, INF, NaN.

The word “approximately” is used above because different conversion routines on different platforms may behave slightly differently. The conversion from `string` to the internal binary representation can handle any precision, but the precision of the result of the conversion cannot exceed the precision of the internal representation. The conversion of the internal representation to `string` also leads to limitations of precision. This implementation defines that the conversion of `float` to `string` returns at most 6 significant digits, and the conversion of `double` to `string` returns at most 14 significant digits.

Conversion to `string` always yields the canonical representation. This applies also to `integer`, `unsigned integer` and `decimal`.

Type Propagations

When two operands in an arithmetic expression or a comparison are of different numeric types, Tamino ensures that the results are correctly evaluated. For example, if you add an `integer` to a `decimal`, the `integer` is converted to `decimal` and the result is calculated as a `decimal`. The conversion of the `integer` to `decimal` may result in an overflow, as the value space of `decimal` is smaller than the value space of `integer`. Overflow may also occur when attempting to convert, for example, a negative `integer` to type `unsigned integer`.

Generally, types are propagated in the following order: `integer` --> `unsigned integer` --> `decimal` --> `float` --> `double`.

Exceptions from this rule

If, for example, an `integer` and a `double` are added, the `integer` is not converted via the steps in the chain. That would cause unnecessary loss of precision. Instead, the `integer` is converted directly to `double`.

If the parameters to the `min()` function are a sequence of integers and unsigned integers, the result is normally `unsigned integer`. However, if at least one sequence member is negative, the result is `integer` (to avoid overflow).

Ordering and Comparison Operations with Datatypes for Date and Time

The following applies for datatypes such as `date` or `time`:

- All datatypes indicating dates or times are partially ordered datatypes. For example, this applies to the following types:
  - `xs:dateTime`
  - `xs:date`
  - `xs:time`
  - `xs:gYear`
  - `xs:gYearMonth`
  - `xs:gMonth`
  - `xs:gMonthDay`
  - `xs:gDay`
  - `xs:duration`
- Comparing two values belonging to these datatypes yields one of three possible results:
  - "true"

- "false"
- "undefined"

Relationship Between XML Schema Types and integer Types

The signed integer type is used for the following **XML Schema** types:

- `long`
- `integer`
- `int`
- `short`
- `byte`
- `negativeInteger`
- `nonPositiveInteger`

The unsigned integer type is used for the following **XML Schema** types:

- `unsignedLong`
- `unsignedInt`
- `unsignedShort`
- `unsignedByte`
- `positiveInteger`
- `nonNegativeInteger`

Defining Simple Types

There is one mechanism for creating a new simple type from an existing base type, namely restriction.

Constraining facets can be used to restrict the value space of an existing simple type (which is specified as the `base` attribute) using the restriction element.

This is done in Tamino in the same way as in the **XML Schema standard**.

The following constraining facets, as defined in the **XML Schema standard**, are available in Tamino:

- `xs:enumeration`
- `xs:fractionDigits`
- `xs:length`
- `xs:maxExclusive`

- xs:maxInclusive
- xs:maxLength
- xs:minExclusive
- xs:minInclusive
- xs:minLength
- xs:pattern
- xs:totalDigits
- xs:whiteSpace

They can be applied individually or in combination. However, not all combinations of restricting facets are valid for all datatypes. See the table of *Valid Combinations of Restricting Facets and Base Datatypes* for details.

For improved reusability, it is possible to use the name attribute to specify a name for a simple type. The definition can subsequently be referenced using this name. This is called a named simple type definition. At the highest level, only named type definitions are allowed; anonymous type definitions are only permitted below the highest level.

The **first example below** shows a named simple type definition; the **second example** shows an anonymous simple type definition:

Example of a simple type definition using a single facet (enumeration)

This example uses the enumeration facet to restrict the value space of the base type **NMTOKEN** to three possible values. It defines a simple type for a datatype for characterizing locomotives of different traction based on a restriction that only the three values “steam”, “diesel” and “electric” are allowed data for the defined type:

```
<xs:simpleType name="traction">
  <xs:restriction base = "xs:NMTOKEN">
    <xs:enumeration value = "steam"/>
    <xs:enumeration value = "diesel"/>
    <xs:enumeration value = "electric"/>
  </xs:restriction>
</xs:simpleType>
```

Example of a simple type definition using multiple facets (totalDigits and fractionDigits)

The SQL datatype numeric(18,5) is expressed using the totalDigits and fractionDigits facets as:

```
<xs:element name = "myDecimal">
  <xs:simpleType>
    <xs:restriction base="xs:decimal">
      <xs:totalDigits value= "18"/>
      <xs:fractionDigits value = "5"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Defining Complex Types

A simple type definition allows neither the definition of a node that contains other elements nor the definition of a node that contains attributes; therefore, the nodes that can be defined using simple type definitions are terminal nodes in the XML tree. To define more complex nodes containing elements or attributes, a more sophisticated kind of type definition is required, namely the complex type definition. It allows the following:

- The definition of attributes using the `xs:attribute` and `xs:anyAttribute` child elements of the `xs:complexType` element;
- Constructs like `choice`, `sequence` and `all` using the `xs:choice`, `xs:sequence` and `xs:all` elements;
- The extension or restriction of an existing datatype, using the definition of either a simple content model (using the `xs:simpleContent` element) or a complex content model (using the `xs:complexContent` element).

This section deals with the following topics:

Definition of Element and Attribute Wildcards

`xs:any` (Definition of Element Wildcards)

The `xs:any` element enables you to extend the instances with elements that are not specified by the schema. It is allowed in `xs:choice` and `xs:sequence` elements.

```
<xs:element name="client">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="cl_firstname" type="xs:string"/>
      <xs:element name="cl_lastname" type="xs:string"/>
      <xs:any minOccurs="0"/>
    </xs:all>
```

```
  </xs:complexType>
</xs:element>
```

`xs:anyAttribute` **(Definition of Attribute Wildcards)**

This element is available for the specification of attribute wildcards in complex type definitions. It allows the occurrence of arbitrary attributes with the current element in the XML instance to be validated against the schema. It is allowed in the `xs:choice` element.

Model Groups: Choices, Sequences and All

`xs:complexType` is used to:

- Define a complex content model using model groups. A model group is composed of particles, which are elements, wildcards (represented by `xs:any`) and nested model groups. A particle has a valid occurrence count, which is determined by the values of the `minOccurs` and `maxOccurs` attributes. The default value of each of these attributes is 1. The following types of model groups exist:

  `xs:choice`
  : `xs:choice` specifies a set of mutually exclusive particles. With this model group you can specify that exactly one of the particles specified in `xs:choice` must occur in the element.

  `xs:sequence`
  : `xs:sequence` specifies an ordered set of particles. All of the particles must occur in the given order in the element.

  `xs:all`
  : `xs:all` specifies an unordered set of elements. All elements specified must occur in the element, but they can occur in any order.

  Model groups themselves can be nested inside an `xs:sequence` or `xs:choice` element.

  Example:

  An element named “address”, which may contain either a postal address or a telephone number or an email address:

```
<xs:element name="address">
  <xs:complexType>
    <xs:choice>
      <xs:sequence>
        <xs:element name="street" type="xs:string"/>
        <xs:element name="zip"    type="xs:integer" minOccurs="0"/>
        <xs:element name="city"   type="xs:string"/>
      </xs:sequence>
      <xs:element name="phone" type="xs:string"/>
      <xs:element name="email" type="xs:string"/>
    </xs:choice>
```

```
  </xs:complexType>
</xs:element>
```

The following fragments validate against this schema fragment:

```
<address>
  <street>5th Avenue</street>
  <city>New York</city>
</address>
```

```
<address>
  <phone>32168</phone>
</address>
```

```
<address>
  <email>E.Hillary@mt-everest.org</email>
</address>
```

whereas

```
<address>
  <street>5th Avenue</street>
  <city>New York</city>
  <email>E.Hillary@mt-everest.org</email>
</address>
```

does not validate.

- Add attribute definitions.

  Example 1:

  An element named `fee` with an `amount` attribute and a `currency` attribute, but empty content:

```
<xs:element name="fee">
  <xs:complexType>
    <xs:attribute name="amount" type="xs:decimal" />
    <xs:attribute name="currency" type="xs:token" />
  </xs:complexType>
</xs:element>
```

  An example of a valid instance:

```
<fee amount="10" currency="USD"/>
```

Example 2:

An element named `fee` containing a decimal number with a `currency` attribute:

```
<xs:element name="fee">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:decimal">
        <xs:attribute name="currency" type="xs:token"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

An example of a valid instance:

```
<fee currency="EUR">10</fee>
```

- Allow for additional more or less arbitrary child elements or attributes using wildcards, i.e. the elements `xs:any` and `xs:anyAttribute` respectively.

  The **XML Schema** for schemas (and thus TSD) allows arbitrary attributes belonging to any other non-XML-schema namespace to be specified in any element. For example, the definition of the `xs:schema` element has the following structure:

  Example:

```
<xs:element name="schema">
  <xs:complexType>

    <xs:sequence>
      ...
    </xs:sequence>

    <xs:attribute name="targetNamespace" type="xs:anyURI" />
    <xs:attribute name="version" type="xs:string" />
    ...
    <xs:anyAttribute namespace="##other" />

  </xs:complexType>
</xs:element>
```

- Allow for mixed content (where both text and child elements are allowed) by using the `mixed="true"` attribute.

Extension and Definition of Simple Content Models

Another possibility for creating a complex type is to define a simple content model. A simple content model for a complex type can be created by extending an existing base type with additional attributes. This is done using the `xs:simpleContent` element and its child element, `xs:extension`.

The following examples illustrate this:

Example 1:

A complex type is defined with a simple content model constructed as follows:

An attribute `duration` of type `unsignedShort` is added to an element of type `normalizedString`

```
<xs:element name="Track">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base = "xs:normalizedString">
        <xs:attribute name = "duration"
                      type = "xs:unsignedShort"
                      use = "required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

Example 2:

The attributes `language` and `length` are added:

```
<xs:element name = "TITLE">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base = "xs:string">
        <xs:attribute name = "language"
                      type = "xs:string"/>
        <xs:attribute name = "length"
                      use = "required"
                      type = "xs:string"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

Example 3:

Perhaps surprisingly, an empty element is also modeled in TSD using an empty complex type definition with the `mixed` attribute set to the value *false*:

```
<xs:element name = "Tag">
  <xs:complexType mixed="false"/>
</xs:element name>
```

Untyped Element

If neither a type attribute nor a simple or complex type is specified for an element, arbitrary attributes and child elements are permitted.

Examples of XML Data Typing

For more information about the Tamino query language, see the X-Query User Guide.

Example: Natively-Stored Doctype with some Formally-Structured Nodes

The following example specifies the "born" node as a field of type `integer` in a doctype stored natively in XML that is to be indexed for full text retrieval. In **XML Schema**, it can be represented by this code:

```
<xs:schema xmlns:xs ="http://www.w3.org/2001/XMLSchema">
  <!-- schema for patient data. This could be a fragment of a larger
       DTD modeling hospital data -->
  <xs:element name='patient'>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref='name' minOccurs='0'/>
        <xs:element ref='address' minOccurs='0'/>
        <xs:element name='born' type='xs:integer'/>
      </xs:sequence>
      <xs:attribute name='ID' type='xs:string' use='optional'/>
    </xs:complexType>
  </xs:element>
  <xs:element name='name'>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref='surname'/>
        <xs:element ref='firstname'/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name='surname' type='xs:string'/>
  <xs:element name='firstname' type='xs:string'/>
  ...
```

```
  <tsd:elementInfo>
    <tsd:physical>
      <tsd:native>
        <tsd:index>
          <tsd:standard/>
        </tsd:index>
      </tsd:native>
    </tsd:physical>
  </tsd:elementInfo>
  ...
</xs:schema>
```

This improves the performance of the execution of requests such as: “List all patients born after a certain date”, for example:

```
...../patient[born >= 1950]
```

Substitution Groups

The motivation for substitution groups originates in the area of object-oriented design. Assume that a schema includes the following global element declarations:

```
<xs:element name="name" type="xs:string" />
<xs:element name="surname" type="xs:string" substitutionGroup="name" />
```

Then the element `<surname>` may replace `<name>` in any context where `<name>` would have been validated against the element declaration shown above. In general, the type of the substituting element must be derived from the type of the substituted element.

Identity Constraints

Identity constraints allow you to define unique constraints or keys that are checked within the scope of a single XML document that is validated against the schema.

As an example, we consider a schema that describes a company and its employees:

```
<xs:element name="company">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element ref="address"/>
      <xs:element ref="employee" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="employee">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="boss" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

We assume that each employee has a unique name. We further assume that, as a rule, an employee has a boss who is also an employee (of the same company). These constraints can be described by the following extension of the element declaration shown above:

```
<xs:element name="company">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element ref="address"/>
      <xs:element ref="employee" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:key name="emplName">
    <xs:selector xpath="employee"/>
    <xs:field xpath="name"/>
  </xs:key>

  <xs:keyref name="emplNameRef" refer="emplName">
    <xs:selector xpath="employee"/>
    <xs:field xpath="boss"/>
  </xs:keyref>

</xs:element>
```

The `<xs:key>` constraint asserts that:

- Each employee's name is unique within the company;
- Each employee has a name;

- An employee cannot have more than one name.

If `<xs:unique>` is used instead of `<xs:key>`, the second condition is not enforced by the identity constraint. Based on the extended declaration of the `company` element, the employee:

```
<employee>
  <name>Bob Smith</name>
  <boss>Alex Miller</boss>
</employee>
```

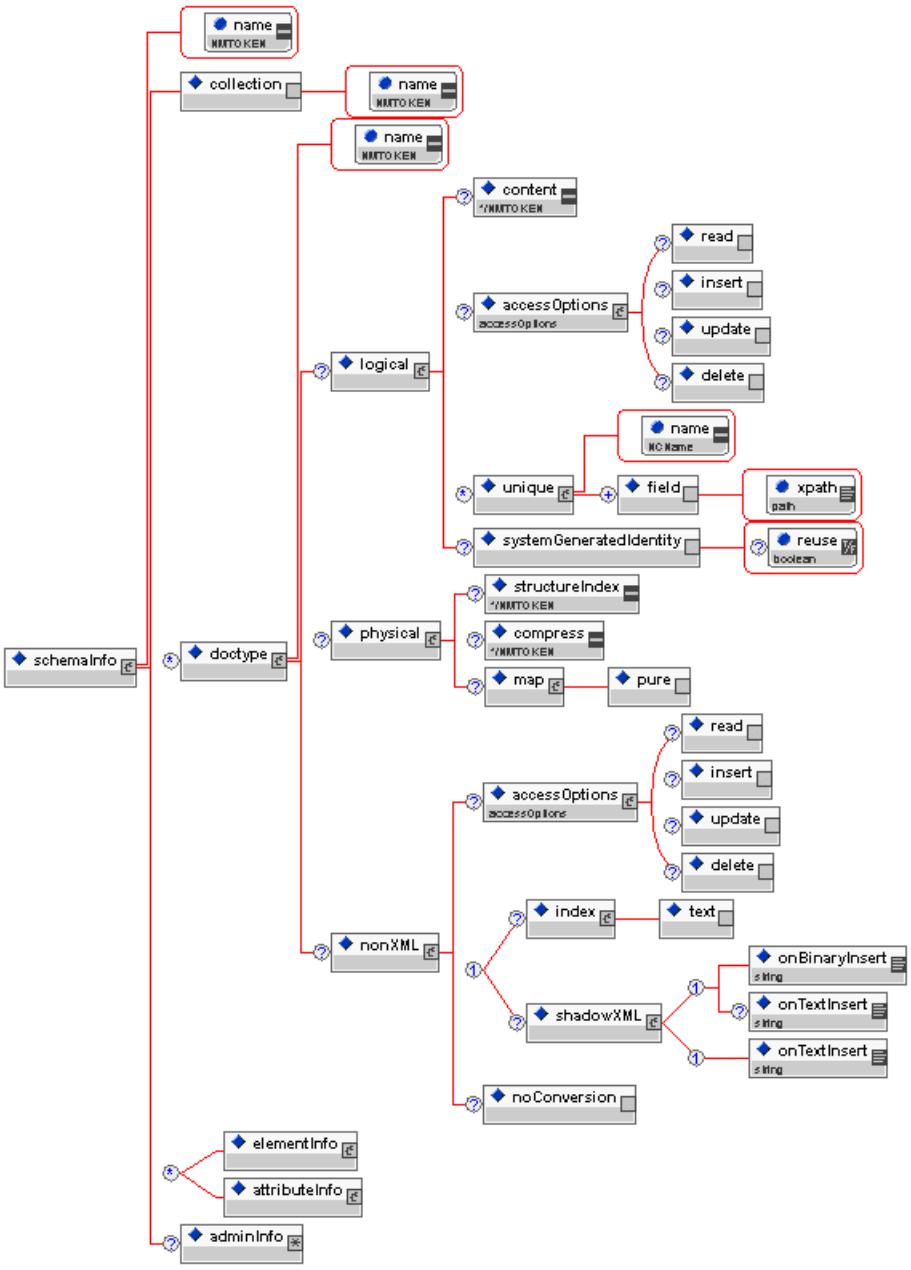
is only valid if:

- There is no other employee named Bob Smith; and
- There is another employee named Alex Miller.

4 The Schema Header

All information relevant to the description of a schema that cannot be expressed in the `xs:` namespace is located in the `tsd:schemaInfo` element of TSD and its subtree. This is also called the "schema header". The `tsd:schemaInfo` element is embedded into the `xs:appinfo` element (this is explained in the section ***The xs:annotation / xs:appinfo Mechanism for Adding Tamino-Specific Extensions***).

The following diagram illustrates the content model of the `tsd:schemaInfo` element used under the `xs:annotation` and `xs:appinfo` elements:



Note: The illustrated `tsd:schemaInfo` element is located under the `xs:appinfo` and `xs:annotation` elements directly under the top-level `xs:schema` element in the XML Schema syntax.

In detail, `tsd:schemaInfo` contains:

- **The `name` attribute.**
  The meaning of this attribute differs slightly depending on its context:

  In the context of `tsd:schemaInfo`
  When used with the `tsd:schemaInfo` element, the value of this attribute is the name of the schema. For a more detailed description, see the documentation of the `tsd:schemaInfo` element.

  In the context of `tsd:collection`
  See below under *The `tsd:collection` Child Element*.

  In the context of `tsd:doctype`
  See below under *The `tsd:doctype` Child Element*.

- **The `tsd:collection` Child Element**
  The `name` attribute of the `tsd:collection` child element specifies the name of the collection to be used in cases where this name cannot be clearly determined. You can also decide at this point whether an explicit schema is required when processing instances or not: see `tsd:schema`.

- **The `tsd:doctype` Child Element**
  The `tsd:doctype` child element specifies the name of the Tamino doctype that the schema describes. It can also specify **logical** and **physical** storage options and special options for **non-XML data**. The doctype-specific logical storage options provided by this element include the choice between storage as **open or closed content**. The **doctype-specific physical storage options** include the possibility of creating a structure index and the "compress" option.

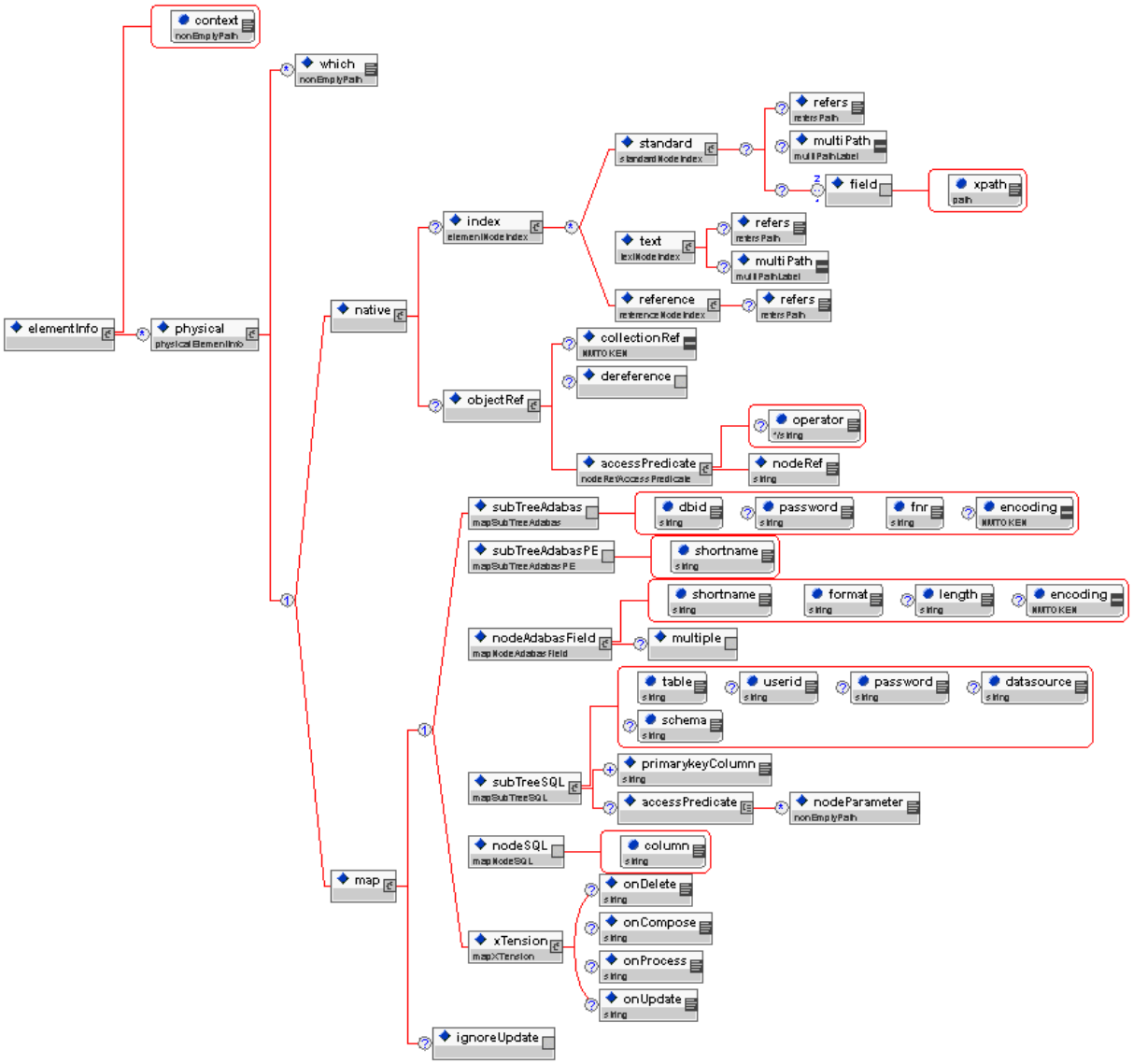
  A single schema can define multiple XML and non-XML doctypes.

- **The `tsd:adminInfo` Child Element**
  The `tsd:adminInfo` child element contains administrative information that is associated with the current schema. This includes version information, creation date and modification date. This information is generated by Tamino when the schema is defined. It does not need to be maintained by the schema author.

- **The `tsd:elementInfo` Child Element**
  The `tsd:elementInfo` child element contains information that describes an element in the sense of **XML Schema**, and also all relevant mapping or indexing information that is attached to this structural element. This includes both logical and physical information:

  The logical information included in the `tsd:elementInfo` element contains information about collation handling relevant to the schema, the definition of action triggers and functions that generate default values. For more information, see the respective sections of this document.

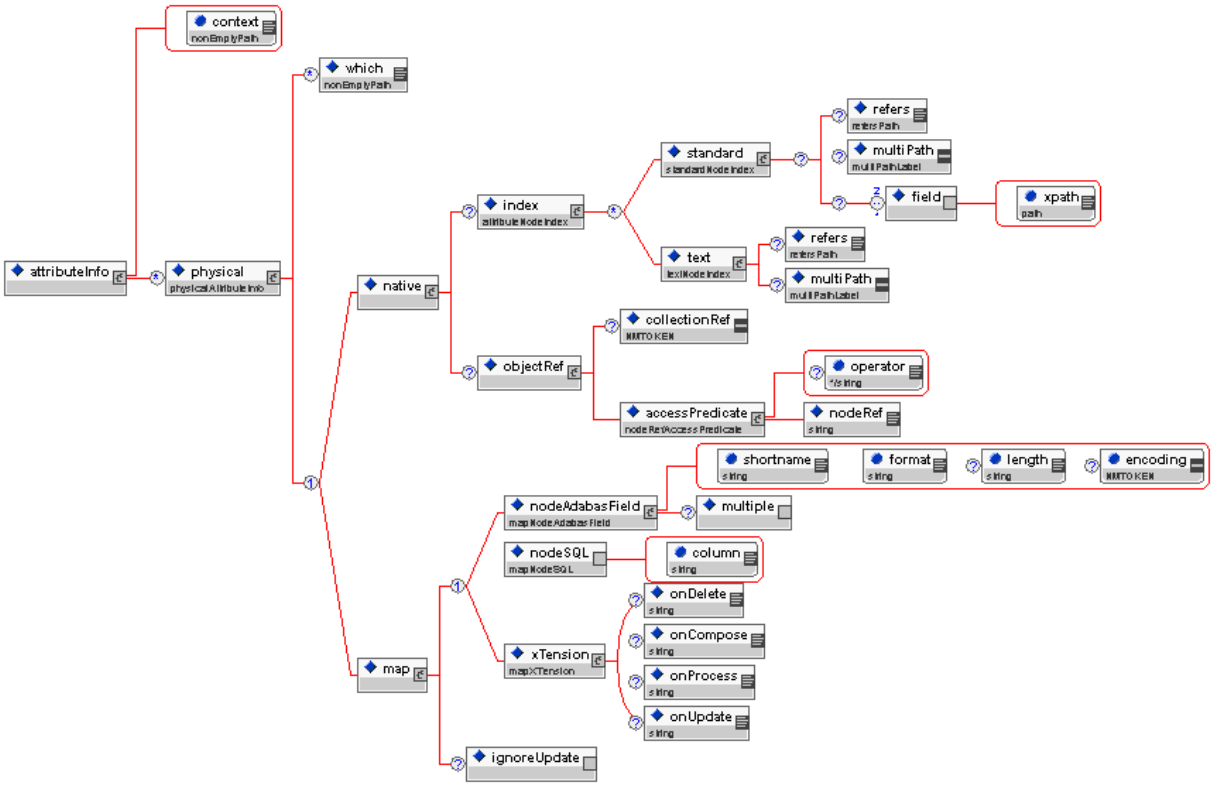
  The physical information relates to indexing, native and external mapping and object referencing, i.e. linking of different elements. In addition, the `tsd:which` element allows you to define an access path in situations where multiple paths are possible. For more information, see the section ***Elements and Attributes*** of this documentation.



- **The `tsd:attributeInfo` Child Element**

  The `tsd:attributeInfo` element contains both logical and physical information that describes an attribute in the sense of **XML Schema**.

  Similarly to the `tsd:elementInfo` element, both logical and physical information is included in the `tsd:attributeInfo` element. The information about mapping, indexing and object referencing given in the section ***Elements and Attributes*** of this documentation also applies to attributes.



Example for a Schema Header

The following example shows a schema for storing data about hospital patients.

Note: The full schema is given here. The start and end of the schema header are marked by comments.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema"
           xmlns:tsd = ↩
"http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
<!-- schema for patient data. This could be a fragment of a larger DTD
     modeling patient data in a hospital-->
<!-- Start of schema Header -->
  <xs:annotation>
    <xs:appinfo>
      <tsd:schemaInfo name = "patient-schema">
        <tsd:collection name = "Hospital"/>
        <tsd:doctype name = "patient"/>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
<!-- End of schema Header -->
```

```
...
</xs:schema>
```

Persistence of Schema Documents

Schema documents defined successfully to Tamino are stored in the doctype `xs:schema` in the collection `ino:collection`. Furthermore, a unique document name is generated for each schema document consisting of the collection name, a slash ("/") and the schema name. Thus, the schema given above could be retrieved from Tamino by using the following plain URL addressing:

```
http://node_name/tamino/dbname/ino:collection/xs:schema/Hospital/patient-schema
```

where *node_name* and *dbname* are placeholders for the nodename and the database name to be used, respectively.

5 Tamino-Specific Extensions to the Logical Schema

Some special aspects of logical features of the Tamino Schema Definition are described in the following sections:

Open Content vs. Closed Content Validation

The validation of instances of a doctype defined in a TSD schema is controlled by the value of the `tsd:content` element in the respective `tsd:doctype` element. This value may be either “closed” or “open”.

Closed content validation means that the instances are validated against the corresponding TSD schema, which is strictly based on the ***XML Schema Recommendation***.

As a special feature, Tamino also enables you to use open content.

Open content validation is defined locally for the complex type definition of one element. For an element "`e`", we define its child-element-name-set (CNS) as the set of names of the children in `e`'s content model (i.e. the names of all elements that might appear as child nodes in "`e`", including globally declared elements that are referenced via an `xs:any` wildcard). Open content validation validates all child nodes in "`e`" that have a name contained in its CNS as usual. All other child nodes are ignored.

For an element `e`, we define its attribute-name-set (ANS) as the set of names of attributes in `e`'s content model (i.e. the names of all attributes that might appear with "`e`", including globally declared attributes that are referenced via an `xs:anyAttribute` wildcard). Open content validation validates all attributes in `e` that have a name contained in its ANS as usual. All other attributes are ignored. For elements with `xs:simpleType`, the open content validation is not changed compared to closed content. It is not possible to define an appropriate validation for mixed content with `xs:simpleType` other than `xs:string`.

Note: The open content validation of attributes is equivalent to the addition of an `<xs:anyAttribute processContents="lax" namespace="##any">` statement to every element. If `xs:anyAttribute` is already specified, the looser definition applies.

Example:

```
<xs:element name = "PLAY">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="THEATRE"
                  type="xs:string"/>
      <xs:element name = "ACT">
        <xs:complexType>
          <xs:sequence>
            <xs:element name = "TITLE" type = "xs:string"/>
            <xs:element name = "COMMENT"
                        type = "xs:string"
                        maxOccurs = "2"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The CNS of `PLAY` is {`THEATRE`, `ACT`}. The CNS of `ACT` is {`TITLE`, `COMMENT`}. The following instance validates, because the elements `MYTITLE` and `THEATRE`, which are not in the content model of `ACT`, are not in the CNS of `ACT`, thus they are ignored during open content validation:

```
<PLAY>
  <THEATRE>aTheater</THEATRE>
  <ACT>
    <MYTITLE>anotherTitle</MYTITLE>
    <TITLE>aTitle</TITLE>
    <THEATRE>anotherTHEATRE</THEATRE>
    <COMMENT>c1</COMMENT>
    <COMMENT>c2</COMMENT>
  </ACT>
</PLAY>
```

The following instance does not validate, because `THEATRE` appears in the CNS of `PLAY` and appears too often in the document. Furthermore, `COMMENT` belongs to the CNS of `ACT` and must not appear before `TITLE` child elements.

```
<PLAY>
  <THEATRE>aTheater</THEATRE>
  <THEATRE>anotherTHEATRE</THEATRE>
  <ACT>
    <MYTITLE>anotherTitle</MYTITLE>
    <COMMENT>c3</COMMENT>
    <TITLE>aTitle</TITLE>
    <THEATRE>anotherTHEATRE</THEATRE>
    <COMMENT>c1</COMMENT>
    <COMMENT>c2</COMMENT>
  </ACT>
</PLAY>
```

Open content validation can be compared to an `xs:any` wildcard or an `xs:anyAttribute` wildcard with `processContents="lax"`. It validates known elements and attributes and accepts unknown elements and attributes. For “lax”, all global elements are known elements, whereas for open content the CNS consists of known elements.

Extensive usage of `xs:any` often leads to ambiguous, i.e. invalid schemas. To circumvent this problem, either do not allow such open content or specify `xs:any` for element content where you do know more details but cannot express this without creating ambiguity. In addition to **XML Schema**'s `xs:any` element, Tamino's open content allows you to describe known schema constraints while at the same time allowing for a priori unknown child elements.

The default value is *closed content* (`<tsd:content>closed</tsd:content>`).

Open content is a feature of Tamino that is neither defined in DTD nor in **XML Schema**, but nevertheless makes working with Tamino easier. In addition, `xs:any` as defined by **XML Schema** is supported. The `xs:any` element is supported with all possible values of the `processContents` attributes, i.e. "skip", "lax" and "strict").

Update Schema Processing for Open Content

Schema evolution (also called “update schema”) for open content is described in the section *Schema Evolution for Open Content (Update Schema Processing)*.

Collations

A collation defines an ordering relationship (less than, equal, greater than) between pairs of characters. It applies when comparing characters or strings; by extension, it also applies when sorting characters or strings. More exactly, the process of arranging elements of a set into a particular order is defined as collation. A collation can also specify the ordering of modified letters, e.g. letters with umlauts that appear in some languages like German or the Scandinavian languages.

Tamino uses the **ICU** (International Components for Unicode) software package for collation.

Extensive documentation on the **ICU** package can be found at **http://www.icu-project.org/docs/**.

In TSD, collations are handled by a `tsd:collation` element, which provides some child elements for specifying Tamino's collation-specific behavior.

Notes:

1. All unspecified collation attributes are set to the default values associated with the specified language/locale.
2. It is also possible to specify no child elements at all, leading to an empty collation element: `<tsd:collation/>`. In this special case, the root locale (in which the pure Unicode Collation Algorithm (UCA) is applied) has to be used.

`tsd:language`
: Specifies information about the language to be used. Typically, you choose a predefined table by selecting a language.

  Note: If no `tsd:language` child element is specified, the root locale is used as default.

`tsd:strength`
: Specifies strength information, i.e. the level of comparison which applies for the collation. (There are five possible levels of comparison for collations.)

`tsd:caseFirst`
: Specifies whether special treatment for uppercase and lowercase characters is intended.

`tsd:alternate`
: Specifies information on punctuation handling in sorting.

`tsd:caseLevel`
: Enables different handling of case levels, see below.

`tsd:french`
: Specifies French accent sorting order information.

`tsd:normalization`
: Specifies information on normalization.

The default values for each collation attribute refer to the root locale, and not the default for any specific locale specified in the `tsd:language` element. For example, if the language is Latvian "lv", the default for `tsd:caseFirst` is "upperfirst", and not "off" (which is the “default” for the root locale which is language-neutral). Another similar language-specific default is the `tsd:french` option that is defaulted to "true" and not "false" if the current language is specified as French ("fr").

The `tsd:collation` element is located in the logical subtree of the `tsd:elementInfo` or `tsd:attributeInfo` element.

tsd:language

The `tsd:language` element specifies the language and the country for which to apply the sorting order table. This is done by assigning a string containing the locale of the according language and country to the value attribute of the `tsd:language` element. Tamino's locale is exactly the **ICU** locale, since Tamino uses the ICU software package.

A locale in the sense of ICU has up to three parts:

1. **ISO 639 Language Code**
   The first argument is a valid ISO language code. These codes are the lowercase two-letter codes as defined by the ISO 639 standard.

   You can find a full list of these codes at **http://www.loc.gov/standards/iso639-2/iso639jac.html**.

   (This standard also defines three-letter codes, which are not applicable in Tamino.)
2. **ISO 3166 Country Code**
   The second argument is a valid ISO country code. These codes are the uppercase two letter codes as defined by the ISO 3166 standard. You can find a full list of these codes at **http://www.iso.org/iso/country_codes.htm**.

   (This standard also defines three-letter codes, which are not applicable in Tamino.)
3. **Suffix**
   A suffix can be specified for various purposes, for example EURO for specifying a table containing a Euro currency symbol (€).

   Note: There is no difference in the sorting order whether EURO is specified or not.

In ICU, a locale is represented simply by a string that consists of a mandatory ISO-639 language code as explained above plus an optional ISO 3166 country code plus an optional suffix. Tamino allows two different styles of syntax:

- **RFC-1766 Style (recommended)**
  The fields are separated by a minus sign. For example, German for Germany with a Euro sign would be expressed as `de-DE-EURO`.
- **ICU Style**
  The fields are separated by an underscore sign. For example, German for Germany with a Euro sign would be expressed as `de_DE_EURO`.

Note: Locale names are case-insensitive. By convention, the language code is lowercase, the country code is uppercase and the variant part is uppercase.

The table listing the values that are available for the `value` attribute of the `tsd:language` element can be found in the **appendix**.

tsd:strength

The `tsd:strength` element has a `value` attribute that specifies the strength information, i.e. the level on which comparison operations are performed. The`value` attribute can have one of 5 possible values:

`primary`
: Level 1: The base characters are compared, e.g., “a” < “b”. No other aspects are taken into account.

`secondary`
: Level 2: Not only the characters themselves are compared, but also accents on characters are compared, e.g. “as” < “às” < “at”.

`tertiary`
: Level 3 (default): Case-sensitive comparison: uppercase and lowercase characters are compared, e.g. “ao” < “Ao” < “aò”. This is ignored if a difference is found on level 1 or level 2.

`quaternary`
: Level 4: Punctuation-sensitive comparison distinguishes words with and without punctuation, e.g. “ab” < “a-b” < “aB”. This is ignored if a difference is found on level 1-3, and should be used only if a distinction based on punctuation is required.

`identical`
: Level 5: This comparison level is used if levels 1-4 would yield identical results. The Unicode code point values are compared. Note that this level of comparison can impact performance negatively.

tsd:caseFirst

If specified, this element makes case the most significant factor when performing collation at the tertiary level (level 3). Collation at primary and secondary levels is unaffected by this option. It has the `value` attribute, which can have one of the following three values:

"**upperFirst**"
With "upperFirst", words starting with uppercase are sorted together before words starting with lowercase.

"**lowerFirst**"
A value of "lowerFirst" causes exactly the opposite behavior.

"**off**"
A value of "off" indicates no distinction is made between uppercase and lowercase during sorting.

The default is "off"

Note: If this option is set to either "upperFirst" or "lowerFirst", words starting with the same case are sorted together either uppercase or lowercase first. Mixed case words (e.g. “AbC”, “aBc”) are therefore always sorted between uppercase and lowercase.

tsd:alternate

This element is used for specifying information on the handling of punctuation in sorting. Its value attribute can take the following values:

- The value “shifted” sorts words containing punctuation marks together (e.g. bi-weekly and bi-weekly). This means that punctuation is ignored for levels 1-3.
- The value “nonIgnorable” (default) distinguishes these words and sorts them separately. This means that punctuation marks are taken into account.

tsd:caseLevel

The `tsd:caseLevel` element has a `value` attribute that can have the value "true" or "false". This option, which is independent of the strength of comparison, introduces an extra level for case differences between secondary and tertiary if set to "true". It can be used in Japanese to make small/large Japanese Kana characters more significant than other tertiary differences. It can also be used to ignore tertiary or secondary differences except for case. For example, if strength is set to "primary" and case level is "true", the comparison ignores accents and other tertiary differences except for case. The default for the value attribute is "false".

tsd:french

This element is used for specifying French accent sorting order information. (French and some other languages require words to be ordered on the secondary level according to the last accent difference.)

Possible values for the `value` attribute are “true” and “false”.

The value “true” produces French accent sorting. The default is “false”, but the function is switched on if the collation language is "*fr*".

tsd:normalization

This element has a value attribute that is used to decide whether or not text normalization is performed.

Possible values for the `value` attribute are “true” and “false”.

The value "true" produces results as if text were normalized. The default is "false", meaning that no normalization is done. Most languages require the value "true" for consistent results.

Storing Non-XML Objects in Tamino

The Tamino Schema Language allows you to define a non-XML doctype. Thus it is possible to store non-XML objects in Tamino, for example, word processor documents, spreadsheets, SGML instances, HTML documents, “broken” (i.e., not well-formed) XML, graphic files, multimedia files, etc.

These objects are stored using HTTP PUT requests or Tamino `_process` commands by referring to the object's database URL (see the URL format for plain URL addressing).

Schema definition for storing and retrieving non-XML objects is described under the following headings:

- **Definition of Non-XML Objects using the `tsd:nonXML` element**
- **Schema and Storage Considerations**
- **Example of Non-XML Storage**

Definition of Non-XML Objects using the tsd:nonXML element

You can write a simple **XML Schema** for the objects to be stored. You can use the Tamino Schema Editor or a text editor to create a Tamino schema, since writing a schema for a non-XML object is usually trivial.

The following is a description of attribute values specific to storing non-XML objects.

`tsd:nonXML` **element**
The `tsd:nonXML` element in the definition of a doctype indicates that data belonging to this defined doctype is stored in Tamino as non-XML data (and not in the standard XML data storage area). This element may only occur within a doctype definition. It excludes all options for defining conditions for XML storage. It offers the possibility to define a text index (see below).

`tsd:index`
In this context, this element is a child element of `tsd:nonXML`. It specifies that a non-XML doctype declared with the containing `tsd:nonXML` element will be indexed with a full text index. For example, the doctype element could look like this if a text index is defined for the specific doctype (only the fragment of the whole schema definition belonging to the doctype element is shown):

```
<tsd:doctype name = "X-Rays">
  <tsd:nonXML>
    <tsd:index>
      <tsd:text/>
    </tsd:index>
  </tsd:nonXML>
</tsd:doctype>
```

Note: A standard indexing option does not make sense in this context and is therefore not available for non-XML data.

Note: A doctype that stores non-XML data has a name just like any other doctype. Unlike a doctype for XML data, no global element whose name corresponds to this is required.

No Conversion for Non-XML Text Documents

By default, Tamino converts all text documents stored in a non-XML document to an internal UNICODE-based encoding. In some scenarios, e.g. for usage with WebDAV, this means that the document returned upon retrieval may not be identical to the document that was originally stored. This can be prevented by using the following schema fragment:

```
<tsd:doctype name="...">
  <tsd:nonXML>
    <tsd:noConversion/>
  </tsd:nonXML>
</tsd:doctype>
```

Schema and Storage Considerations

You could theoretically specify a single schema to describe all the non-XML objects you wish to store in Tamino. In practice, however, you may wish to differentiate between types of objects and/or object domains. For example, if patients' X-rays are to be stored electronically in Tamino, but their format can differ (for example, GIF or JPG), you can define a doctype “X-Rays” to store any instance.

Note: If you want to verify this, use plain HTTP requests or the Tamino X-Plorer, for example.

To store a non-XML object using the Tamino Interactive Interface

1 Set the Tamino database URL to the database. Enter the following in the `collection` field:

`collection`/*`doctype`*/*`filename`*

2 With the cursor in the **Process** field of the Tamino Interactive Interface, browse to the object you wish to load and choose **Process**.

To retrieve a non-XML object using the Tamino Interactive Interface

1 Set the Tamino database URL to the full path of the target as follows:

http://localhost/tamino/*`database`*/*`collection`*/*`doctype`*/*`filename`*

2 Type an asterisk "*" in the X-Query field and choose **Query** to retrieve the object.

Example of Non-XML Storage

The following example illustrates a schema for storing non-XML objects in Tamino. The objective is to store X-ray pictures electronically in the Hospital collection.

In the TSD, this can be defined by the following XML document fragment:

```
<xs:schema
  xmlns:xs = "http://www.w3.org/2001/XMLSchema"
  xmlns:tsd = "http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
  <xs:annotation>
    <xs:appinfo>
      <tsd:schemaInfo name = "nonXML">
        <tsd:collection name = "Hospital">
        </tsd:collection>
        <tsd:doctype name = "X-Rays">
          <tsd:nonXML>
          </tsd:nonXML>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
</xs:schema>
```

Using HTTP:

To retrieve a single instance by file name (*xatkins.gif*, for example) from the collection "*hospital*" in the Tamino database "*ino*":

```
http://hostname/tamino/ino/hospital/X-Rays/xatkins.gif
```

To retrieve a single instance by object ID (4711, for example) from the collection "*Hospital*" in the Tamino database “ino”:

```
http://hostname/Tamino/ino/hospital/X-Rays/@4711
```

Using Shadow Functions

This section discusses the technique of defining shadow functions under the following topics:

- Introduction
- Schema Definition for Shadow Functions
- Storing the Shadow Document Without the Non-XML Data
- Selecting Text or Binary Function
- Working with Shadow Functions
- Example
- Indexing of Non-XML Data - Shadow Functions vs. Non-XML Text Indexing

- Tamino Non-XML Indexer

Introduction

A shadow function creates a shadow document for a non-XML document. The non-XML document can be stored in Tamino or outside Tamino. A shadow document is an XML document that can contain information such as metadata, index values and other generated values for the corresponding non-XML document. The purpose of the shadow document is to store information that can be used for queries that would not be possible on the original document.

In order to gain maximum benefit from these techniques, it is necessary to apply suitably designed shadow functions.

The schema that defines the non-XML document must also contain appropriate statements to indicate that a shadow function will be used. The next section explains how to do this.

Schema Definition for Shadow Functions

To indicate that a doctype is intended to store non-XML data, add a `tsd:nonXML` element to the `tsd:doctype` element below `tsd:schemaInfo`.

In order to specify which server extension will act as a shadow function, code a `tsd:shadowXML` element as a child of `tsd:doctype`.

Note: This is analogous to the `tsd:index` child element, which allows you to specify information on indexing options.

As the requirements for shadow functions can be quite different depending on the kind of data to be processed, separate shadow functions can be established that cater either for text data or for binary data. These shadow functions can be specified using the child elements of the `tsd:shadowXML` element:

- In order to define a binary shadow function, the `tsd:shadowXML` element must have a `tsd:onBinaryInsert` element specifying the name of the shadow function that processes binary data.
- In order to define a text-based shadow function, the `tsd:shadowXML` element must have a `tsd:onTextInsert` element specifying the name of the shadow function that processes text data.

Notes:

1. One or both of the child elements `tsd:onBinaryInsert` or `tsd:onTextInsert` must be present in the schema in order to define a valid `tsd:shadowXML` element.
2. The name of the Server Extension acting as a shadow function is specified simply as the content of the respective child element.

Storing the Shadow Document Without the Non-XML Data

It is possible to store the shadow document in Tamino without storing the original non-XML document in Tamino. The non-XML document can reside externally, for example in the file system. This allows applications to retrieve indexing information from the shadow document without the requirement to have the non-XML document itself stored in Tamino. Requests to return the non-XML document itself (for example, using plain URL addressing) rather than indexing information about the non-XML document will fail with an HTTP status 404 (file not found).

This feature is activated by the `tsd:storeShadowOnly` element in the definition of the shadow document.

Selecting Text or Binary Function

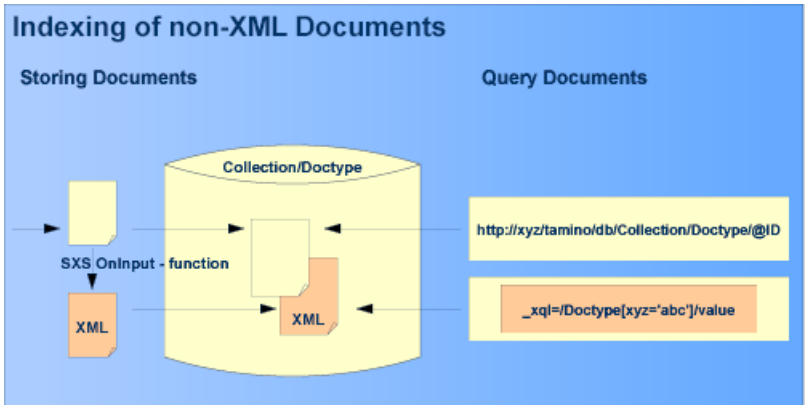
If both a binary shadow function and a text-based shadow function are present, the appropriate function is executed as determined by the criteria described in the section *Media Type Requirements* of the *X-Machine Programming* documentation.

Working with Shadow Functions

The shadow function must be implemented by the user as a Tamino *Server Extension*. It is called when the non-XML document is stored. If `tsd:storeShadowOnly` is defined in the schema, the shadow document is stored in Tamino but the non-XML document is not copied into Tamino.

By calling the shadow function, the user can create an XML document containing values derived from the non-XML document. The XML document is then stored in the Tamino X-Machine as the shadow document that is associated with the non-XML document.

The shadow document can be queried using the XQuery or X-Query query language syntax, just like an ordinary XML document. In fact, all XQuery or X-Query requests will deliver the content of the shadow document instead of the original non-XML document's content.



When a non-XML document is processed, it is passed as a parameter of a suitable binary or textual datatype to the shadow function, which creates the shadow document. The X-Machine inserts the shadow document and performs index processing on it.

The result is stored as a "shadow" of the original non-XML document node in the Tamino server.

We now consider what happens in the following three situations:

- **Document Retrieval**
  As discussed and shown above, all queries issued in Tamino's query languages X-Query and XQuery deliver the shadow document and not the original non-XML document.

  In the case of direct access via plain URL addressing, the non-XML original document is delivered instead of the shadow document. If, however, the option `tsd:storeShadowOnly` is used in the schema definition, the non-XML document is not stored in Tamino, so an HTTP status 404 (file not found) is returned instead.
- **Deletion of the Original Non-XML Document that is stored in Tamino**
  Deleting the instance in Tamino deletes the shadow document as well.
- **Updating the Original Non-XML Document that is stored in Tamino**
  Updating the instance in Tamino deletes the shadow document and re-creates it by executing the shadow function on the updated non-XML document.

  Note: Updating based on XQuery is not permitted.

Example

The following schema fragment shows how to define support for shadow functions within a Tamino schema defining shadow functions named `SXSBinaryIndexer.put` for binary data and `SXSTextIndexer.put` for text data:

```
<?xml version="1.0" encoding="windows-1252" ?>
<xs:schema
 elementFormDefault="qualified"
 xmlns:tsd="http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:annotation>
    <xs:appinfo>
      <tsd:schemaInfo name="nox">
        <tsd:collection name="nox" />
        <tsd:doctype name="nox ">
          <tsd:nonXML>
            <tsd:shadowXML>
              <tsd:onBinaryInsert>
                SXSBinaryIndexer.put
              </tsd:onBinaryInsert>
```

```
              <tsd:onTextInsert>
                SXSTextIndexer.put
              </tsd:onTextInsert>
            </tsd:shadowXML>
          </tsd:nonXML>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>

  <xs:element name="blob">
  .
  .
  .
  </xs:element>
.
.
.

</xs:schema>
```

Indexing of Non-XML Data - Shadow Functions vs. Non-XML Text Indexing

Indexing of non-XML data differs significantly from the usual indexing of XML data as described in the section *Indexing XML Data for Native Storage*. One possibility is to define a text index based on classic full-text search algorithms, as described in the section *Built-In Indexing of Non-XML Data* of this document.

However, this approach is not always appropriate:

- This approach is only applicable when working with text data, not with binary data. In this case shadow functions provide the only possibility to define an index for binary non-XML data.
- If storing non-XML documents, it is not possible to search on content or metadata of such a document other than the docname.

  This happens, for example, if a user wants to store and index office documents (e.g. XML documents created by Microsoft Office programs, HTML or images) and wants to retrieve the documents on behalf of an index document, which should already be created when the document is stored.

Alternatively, you can avoid these disadvantages by using shadow functions to create index values for non-XML documents.

Tamino Non-XML Indexer

The Tamino non-XML indexing software tool is an exemplary application of shadow functions.

The Tamino Non-XML Indexer extends Tamino's facilities for storing and retrieving non-XML objects. It seamlessly integrates non-XML files such as Microsoft Office documents or Star Office documents into the Tamino environment. It enables meaningful searches on the content and/or metadata (such as the date when the document was last changed, the author, etc.) of such non-XML files.

Triggers

XML documents or elements of them can also be associated with a function by the usage of database triggers. This is quite similar to the situation when using shadow functions, also trigger functions can be initiated by the occurrence of a stimulating event. Similarly, the sub-tree of the document to be processed must be associated with the schema in order to indicate which part of the document is affected. This is described below.

The events that can initiate execution of a trigger are:

- Insert
- Update
- Delete

Triggers and trigger functions are described in more detail in *Tamino Server Extensions documentation*.

The `tsd:trigger` element contains all information for logical nodes in the schema that define action triggers for the Tamino *Server Extensions*.

Notes:

1. The Server Extension package that contains the Server Extensions required by your schema definitions must be installed in a Tamino database in order to make the trigger functionality available.
2. *Update* and *Delete* triggers with errors cause the current request to fail and thus prevent the entire document from being stored or deleted successfully.

Depending on the kind of event being processed, three different kinds of trigger functions are currently supported:

Insert functions

For storage or processing: the “Insert” functions/triggers, which are executed before the validation phase during processing.

Update functions
For updating: the “Update” functions/triggers, which are executed before the validation phase during the updating of an XML document.

Delete functions
For deletion: the “Delete” functions/triggers, which are executed when an XML document is to be deleted from the database.

All these are supported by Tamino, depending on the child elements of the `tsd:trigger` element:

- In order to define insert trigger functions, the `tsd:trigger` element must have a `tsd:onInsert` child element specifying the name of the trigger function (Server Extension) that caters for insertions.
- In order to define update trigger functions, the `tsd:trigger` element must have a `tsd:onUpdate` child element specifying the name of the trigger function (Server Extension) that caters for updates.
- In order to define delete trigger functions, the `tsd:trigger` element must have a `tsd:onDelete` child element specifying the name of the trigger function (Server Extension) that caters for deletions.

In each of these cases, the name of the trigger function (Server Extension) that caters for the insertions, updates or deletions is specified simply as the content of the respective element.

Notes:

1. It is possible to specify more than one of `tsd:onInsert`, `tsd:onUpdate` and `tsd:onDelete`.
2. The `tsd:onInsert`, `tsd:onUpdate` and `tsd:onDelete` child elements of the `tsd:trigger` element each have a required attribute `type`. Currently the only permitted value to be specified for this attribute is "action".

Example

The following schema fragment defines trigger functions named `sxsDelete` for delete events, `sxsUpdate` for update events and `sxsInsert` for insert events:

```
<xs:element name = "firstname" type="xs:string">
  <xs:annotation>
    <xs:appinfo>
      <tsd:elementInfo>
        <tsd:logical>
          <tsd:trigger>
            <tsd:onDelete type="action">sxsDelete</tsd:onDelete>
            <tsd:onUpdate type="action">sxsUpdate</tsd:onUpdate>
            <tsd:onInsert type="action">sxsInsert</tsd:onInsert>
          </tsd:trigger>
        </tsd:logical>
```

```
      </tsd:elementInfo
    </xs:appinfo>
  </xs:annotation>
</xs:element>
.
.
.
```

Triggers with Parameters

A trigger function may have an arbitrary number of parameters. A single trigger function, which is implemented as a server extension, can be used in different schemas with a different set of parameters determined by the signature of the underlying server extension function. In addition, you can define an arbitrary number of trigger functions of the same type on one logical node (that is, element or attribute).

Sample Schema

```
.
.
.
<xs:element name="e1" type="xs:string">
  <xs:annotation>
    <xs:appinfo>
      <tsd:elementInfo>
        <tsd:logical>
          <tsd:trigger>
            <tsd:onInsert type="action" name="myXtension.onInsert">
              <tsd:parameters>
                <tsd:parameter name="param1" ↩
type="xs:string">MyClassName</tsd:parameter>
                <tsd:parameter name="param2" type="xs:int">4711</tsd:parameter>
              </tsd:parameters>
            </tsd:onInsert>
          </tsd:trigger>
        </tsd:logical>
      </tsd:elementInfo>
    </xs:appinfo>
  </xs:annotation>
</xs:element>
.
.
.
```

The name of the trigger function should be specified in the `name` attribute of the `tsd:onInsert`, `tsd:onUpdate` or `tsd:onDelete` element. For backward compatibility with older versions of Tamino, it can alternatively be specified as a text node within the `tsd:onInsert`, `tsd:onUpdate` or `tsd:onDelete` element. This facility may be withdrawn in future versions of Tamino.

Signature of Server Extension Function

The signature of the server extension function is the union of:

- the base signature, which depends on the type of the trigger (`onInsert`, `onUpdate` or `onDelete`;
- additional parameters that match the parameters specified in the schema document.

Determining Default Values by Function

Another situation where XML documents or elements of them can be associated with a function is the usage of functions for determination of default values of elements or attributes.

The `tsd:function` child element of the `tsd:default` element allows you to specify a function whose return value defines the default value for an element or attribute within the schema.

Notes:

1. The element containing the function in its definition must not have a `default` or `fixed` attribute.
2. This function is declared in its own namespace.

The following example explains the use of the `tsd:default` element and its child element, the `tsd:function` element, in order to define a function `createDefaultString` in the namespace `company`:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema
 xmlns:tsd = "http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition"
 xmlns:xs = "http://www.w3.org/2001/XMLSchema">
  <xs:annotation>
    <xs:appinfo>
      <tsd:schemaInfo name = "person">
        <tsd:collection name = "person"></tsd:collection>
        <tsd:doctype name = "person"/>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name = "person">
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "name" type = "xs:string">
          <xs:annotation>
            <xs:appinfo>
              <tsd:logical>
                <tsd:default>
                    <tsd:function xmlns:company="http://www.company.com/functions"
```

```
                                    name="company:createDefaultString"/>
                </tsd:default>
              </tsd:logical>
            </xs:appinfo>
          </xs:annotation>
        </xs:element>
        <xs:element name = "first" type = "xs:string">
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Instances Without a Defined Schema

In Tamino, it is possible to store instances without previously defining a schema. There are two ways to store data in a collection without having to previously define a schema:

- Using the Special Collection ino:etc
- Using User-Defined Collections for Storing Instances Without a Defined Schema

Using the Special Collection ino:etc

There is a special collection reserved in Tamino for the storage of instances that have not been supplied with a valid schema. This is the collection `ino:etc`, which is also used if no collection is specified in the `_process` command.

Note: It is recommended not to use `ino:etc` for storing real XML data for which a schema exists and which could be stored in other collections. This kind of usage of `ino:etc` may lead to severe problems, especially with respect to performance, since `ino:etc` was not designed to store real XML data.

Using User-Defined Collections for Storing Instances Without a Defined Schema

Since version 4.2 of Tamino, there is an alternative way of storing data without a previously defined schema which even allows data to be stored in collections with arbitrary names. A collection can be defined so that it only accepts data complying with a previously defined schema that has explicitly been specified (the explicit case), or to accept data also without such a schema (the implicit case); both cases may apply simultaneously. In the implicit case, the system automatically generates a suitable schema and uses it internally.

In detail, this works in the following manner, depending on whether an XML or a non-XML document is affected:

- A `_process` of an XML document into a “schemaless” collection implicitly creates the respective doctype via an internal, hidden schema.

  The XML doctype has a text index on the root node.
- A `_process` of a non-XML document into a “schemaless” collection stores it in the doctype `ino:nonXML`.

This feature is controlled by the `use` attribute of the `tsd:schema` child element of the `tsd:collection` element. This attribute can have the following values:

"**required**"
: This choice reproduces the behavior of the collections of Tamino versions up to 4.1: it is not permitted to store data without having previously defined a schema.

  This is the default value for this option.

"**optional**"
: Both explicit and implicit creation of doctypes are permitted in the respective collection. If a schema is used in the `_define` operation, then the explicit case occurs, otherwise a schema is generated implicitly.

"**prohibited**"
: Only implicit creation of doctypes is possible within the respective collection. It is not possible to perform “normal” schema definition within this collection.

Note: In this case, the `_define` command is used as follows: A collection can be defined by itself as an object carrying properties. This is done via a `_define` command. However, the input for the `_define` is not really an XML schema. This is called “collection definition document”.

Reuse of ino:id

Tamino offers a switch to control at the doctype level whether the reuse of `ino:id`'s is permitted.

In the `tsd:logical` element you can define a `tsd:systemGeneratedIdentity` element, which acts as a switch depending on the value of its `reuse` attribute. The `reuse` attribute may have the value "true" or "false".

- If `reuse="true"`, reuse of `ino:id`'s is possible when inserting documents after deleting other documents. This is the default value.
- If `reuse="false"`, reuse of `ino:id`'s is prohibited.

The example schema below prohibits reuse of `ino:id`'s for doctype A.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:tsd="http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
  <xs:annotation>
    <xs:appinfo>
      <tsd:schemaInfo name="Reusage-Schema">
        <tsd:collection name="Reusage"/>
        <tsd:doctype name="A">
          <tsd:logical>
            <tsd:systemGeneratedIdentity reuse="false"/>
          </tsd:logical>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>

  <xs:element name="A"/>
<xs:schema>
```

6 Tamino-Specific Extensions to the Physical Schema

The physical schema specifies physical storage information, for example for mapping and indexing.

This section describes the Tamino-specific extensions to the physical schema under the following headings:

Doctype Specific Extensions

The following topics are covered in this section:

Structure Index

This element specifies whether a structure index is created for the corresponding doctype. The element is of type `xs:NMTOKEN`. The `tsd:structureIndex` element can have one of the following values:

none
No structural indexing is done.

condensed
The repository registers the existence of an undeclared node for the doctype.

full
The repository registers both the existence of undeclared nodes and the instances in which they occur.

The default value is `condensed`.

Computed Indexes

Computed indexes defined for a doctype allow for user-defined indexing based on an indexing function defined in an XQuery module. For details refer to the section Advanced Indexes in the *Performance Guide*.

Definition of Unique Keys

For a description of how to define unique keys, refer to the topic **Definition of Unique Keys** in the section **Physical Schema for Elements and Attributes**.

Compression

The `tsd:compress` element specifies compression options for the physical storage of the data. It specifies whether instances belonging to the corresponding doctype are physically stored in a compressed or an uncompressed format (the latter is recommended for small data records only). Also, the user can specify maximum compression or optimal performance. The `tsd:compress` element is of type `xs:NMTOKEN`.

Note: This option is not applicable for non-XML data.

The following values are allowed:

smart
This is the default value. Tamino checks the data to be stored and uses the best compromise between speed and size.

always
Always compress as much as possible. This choice is appropriate if you are primarily interested in reducing storage size. Especially for small documents, this minimizes disk space but increases retrieval time.

none
Do not compress small data records. Large documents are not affected by this setting. This setting is recommended if you expect most of your documents to be small (< 8000 characters) and you want to optimize processing speed, at the price of increased storage space.

off
Do no compression at all.

utf8
Each character is replaced by its UTF-8 representation. This can result in a compression factor of up to 4, depending on platform and data.

Built-In Indexing of Non-XML Data

Indexing of non-XML data is possible, but in a somewhat different manner than the indexing of XML data. The following example shows how to define a non-XML index using the `tsd:nonXML` element:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema
 xmlns:xs = "http://www.w3.org/2001/XMLSchema"
 xmlns:tsd = "http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
  <xs:annotation>
    <xs:appinfo>
      <tsd:schemaInfo name = "nonXML">
        <tsd:collection name = "Hospital"></tsd:collection>
        <tsd:doctype name = "X-Rays">
          <tsd:nonXML>
            <tsd:index>
              <tsd:text></tsd:text>
            </tsd:index>
          </tsd:nonXML>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
  .
  .
  .
</xs:schema>
```

A plain text index is created for non-XML character data.

Physical Schema for Elements and Attributes

This section deals with the following topics:

- Attaching Physical Schema Information to Specific Nodes
- Definition of Unique Keys
- Indexing XML Data for Native Storage

The figures in ***The Schema Header*** show those parts of the metaschema describing information that can be associated with elements and attributes via `xs:annotation` and `xs:appinfo` using `tsd:elementInfo` and `tsd:attributeInfo`, respectively:

The meaning and usage of these extension elements and attributes are explained in detail in the *Tamino XML Schema Reference Guide*. A general explanation of the `tsd:elementInfo`/`tsd:attributeInfo` subtree follows.

Attaching Physical Schema Information to Specific Nodes

tsd:which

The `tsd:which` element enables you to specify different physical schema information represented by different `tsd:physical` elements inside a single `tsd:elementInfo` or `tsd:attributeInfo` (i.e. an element or attribute), if a node can be reached via multiple paths because of references to global elements or attributes. Physical schemas for different absolute path expressions can be grouped together. Therefore, if necessary, multiple `tsd:which` elements are allowed in one `tsd:physical` element to specify different possible access paths. If no `tsd:which` element is specified within a `tsd:physical` element, all possible paths not explicitly given in any `tsd:which` within a sibling `tsd:physical` have the same physical schema information.

The `tsd:which` element contains a string of type `xs:string` that describes an **XPath** expression, which must match the following constraints:

- It must be an absolute path expression of the following form:

  ```
  /doctypeName/element/.../{currentElementName | @ currentAttributeName }
  ```

  The **XPath** expression must refer to an element or attribute defined in the schema (including the imported schema information).
- Each **XPath** expression of each `tsd:physical` child of one element must be unique.
- At most one `tsd:physical` without a `tsd:which` element is allowed.

The following rules apply for the default behavior of the `tsd:which` element:

- The default for a non-recursive element is all absolute paths to this element.
- If no `tsd:which` element is specified, the default applies.
- For a recursive element, the following applies: If `tsd:which` is applied with the default option, it relates to the first stage of recursion.
- Attachment of physical schema definition is not possible without the `tsd:which` element below recursive elements.

One situation where the usage of the `tsd:which` element is very advantageous is **multi-path indexing**. You can find an explained example including usage of the `tsd:which` element **here**.

context

The `context` attribute is used in `tsd:elementInfo` or `tsd:attributeInfo` elements specified within `tsd:schemaInfo`. Similar to `tsd:which`, its value is a path that determines the `xs:element` or `xs:attribute` node to which the `tsd:elementInfo` or `tsd:attributeInfo` element belongs.

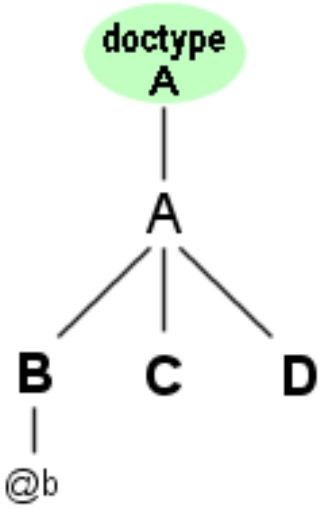
Definition of Unique Keys

Starting with version 4.2, Tamino allows you to specify a uniqueness constraint at the doctype level. This means that you can indicate, during schema definition, fields or combinations of fields that Tamino should monitor, ensuring that they have a unique value within their doctype. The necessary checks are performed automatically when data are inserted and updated (or update-defined). An error message is issued if an attempt is made to violate a uniqueness constraint.

Unique keys are defined centrally in `tsd:schemaInfo/tsd:doctype/tsd:logical`. Below `tsd:logical`, you can define `tsd:unique` elements having one or more `tsd:field` children. These `tsd:field` children allow you to specify how the uniqueness constraint is composed of one or more fields. Each `tsd:field` child element has an attribute `xpath` of type `xs:string`.

Note: The path is relative to the root element.

Caution: If there are multiple `tsd:field` child elements of a given `tsd:unique` element, each child's `xpath` attribute must be distinct.



Based on that, the following schema fragment shows how to define two unique keys for the structure displayed in the diagram:

```
<tsd:doctype name="A">
  .
  .
  .
  <tsd:unique name="CB-key">
    <tsd:field xpath="C" />
    <tsd:field xpath="B/@b" />
  </tsd:unique>
  <tsd:unique name="D-key">
    <tsd:field xpath="D" />
  </tsd:unique>
  .
  .
  .
</tsd:doctype>
```

In the example, two uniqueness constraints are defined: one for the combined key comprising the element `C` and the attribute `b` of element `B`; and the other one for element `D`.

Note: You can define an arbitrary number of `tsd:unique` elements below the `tsd:logical` element.

The names of the fields to which the uniqueness constraint applies are specified using the `xpath` attribute of the `tsd:field` element. This attribute is mandatory. It is of type `xs:string`, with the additional requirement that its value must be a valid **XPath** expression, i.e. it must comply with the ***XML Path Language (XPath) Version 1.0 specification*** published by the **W3C**.

Valid `xpath` attributes are:

- Any **XPath** expression that uses only the axes “child” and “attribute” and does not have more than one attribute;
- The special case of a single dot “.”, which denotes the contents of the root element as a unique constraint.

The following rules apply for unique key definitions:

- **Order of Components**
  The order of components is not significant.

  Example: The following unique key definitions are equivalent:

```
<tsd:doctype name="A">
  .
  .
  .
  <tsd:unique name="CD-key">
    <tsd:field xpath="C"/>
    <tsd:field xpath="D"/>
  </tsd:unique>
  .
  .
  .
</tsd:doctype>
```

and:

```
<tsd:doctype name="A">
  .
  .
  .
  <tsd:unique name="CD-key">
    <tsd:field xpath="D"/>
    <tsd:field xpath="C"/>
  </tsd:unique>
  .
  .
  .
</tsd:doctype>
```

- **Number of Possible Uniqueness Constraints per Doctype**
  There is no theoretical limit to the number of uniqueness constraints that can be specified for one doctype.
- **Condition for Constraint Names**
  Each constraint name must be unique within the doctype.
- **Condition for Number of Components of Constraint**
  Each constraint must have one or more components. Empty constraints are not allowed
- **Exclusion of Double Declarations**
  In any given constraint, the same component may not be used twice.
- **Mixing Elements and Attributes**
  Elements and attributes may be mixed.
- **Mixing Datatypes**
  Different datatypes may be mixed within one single unique key definition.

- **Mixing Collations**
  The mixing of different collations within one single unique key definition is not supported.
- **Condition for Multiplicity of a Component**
  Currently, each component of a unique key must have multiplicity of exactly 1. Further restrictions result from this constraint:
  - **Variety `list` Not Allowed**
    A uniqueness constraint must not reference a node of simple type with variety "list". This includes the predefined datatypes `xs:ENTITIES`, `xs:IDREFS` and `xs:NMTOKENS`.
  - **Reference to Nillable Elements**
    A uniqueness constraint must not refer to an element with `nillable="true"`.
- **Variety `union` Not Allowed**
  A uniqueness constraint must not reference a node of simple type with variety "union".
- **Condition for XPath Expressions**
  XPath expressions are relative to the root element.
- **Addressing the Root Element**
  The expression `xpath="."` refers to the root element.
- **XPath Expressions and Mapped Nodes**
  XPath expressions must not point to nodes with Adabas, SQL or SXS mapping.

  For correct operation of the unique key constraints, all relevant nodes must be updated under the control of Tamino. This is true for all natively-stored data, but not for data under the control of an external system, nor for mapped data whose update cannot be fully controlled by Tamino.
- **Unique Key Definitions and Compound Index Definitions on Identical Fields**
  There is no interaction between unique key definitions and compound index definitions (which are technically related and are discussed **later in this document**). You can define a unique key constraint and a compound index that are based on the same fields. It does not matter whether or not the fields appear in the same order.
- **Criterion for Identity of Extremely Long Keys**
  In order to guarantee uniqueness, Tamino uses an index to keep track of all keys. Tamino truncates keys if they are too long. Currently, the length limit of a key in Tamino's unique key storage area is 1,004 bytes in UTF-8 representation. As a consequence, two keys are considered to be identical if the first 1,004 UTF-8 characters of both keys are identical.
- **Date/Time Types Not Allowed**
  Unique constraints may not be defined on elements or attributes of the following types:
  - `xs:duration`
  - Any datatype that allows an optional timezone indicator, e.g. `xs:dateTime`.

For more information about how to achieve the best performance when using the unique key feature, see the *Performance Guide*.

Indexing XML Data for Native Storage

This section discusses some aspects of the various kinds of indices that Tamino offers and of indexing in XML databases in general. In detail, the following topics are discussed here:

Why Define an Index?

One of the most important reasons for defining a schema for XML data is to index the data for "native"storage, i.e. storing the data in Tamino's internal XML store, although a schema may also be needed for validation and extension purposes.

The performance of database systems depends largely on correct indexing. This is just as true for Tamino as it is for other database systems. Correct indexing is very important, as it is a prerequisite for efficient database processing and retrieval.

However, if you have data for native storage for which no index is required, it is not necessary to make any descriptions, declarations or specifications at all, as `native` is the default storage option for XML data in Tamino, and no indexing is also the default.

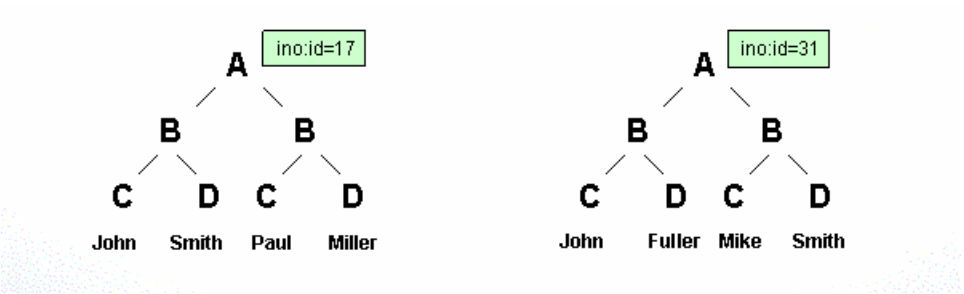
What is an Index?

In general, an index is a look-up table for reducing the time taken to query data stored in a database. The look-up table usually stores the corresponding record number for each occurrence of a given value in a database field. It is possible to define an index:

- on a single field;
- on a combination of fields.

Example

For example, assume a document with the structure displayed in the picture below: a root element `A` has multiple children `B`, each having two children named `C` and `D`. Also assume that there are two instances of this document with the ino:id values 17 and 31 containing the names given in the lowest row of the picture.



Then the index look-up table for `A/B/C` based on these instances would look like this:

John	17, 31
Mike	31
Paul	17

Index for A/B/C

Similarly, the look-up table for an index for `A/B/D` would look like the following:

Fuller	31
Miller	17
Smith	17, 31

Index for A/B/D

However, Tamino does not use record numbers; rather, it uses the `ino:id` to identify data within the database.

Tamino also offers more complex types of indices: see the following sections.

Index Categories Supported by Tamino

There are two ways of classifying indices in Tamino:

- Firstly, indices can be classified according to the search technology on which they are based. From this point of view, we distinguish between text and standard indices.

  Standard index

  A standard index is the classical database index. It is based on standard database indexing technology and is usually datatype independent.

  The complete node content is used as the index value. When a document is stored or updated, Tamino puts data into a specific index, thus enabling efficient queries.

  Tamino offers standard indexing support for all available **XML Schema datatypes**.

Text index
In contrast to a standard index, a text index uses special full-text searching technology based on a word-by-word analysis of text data. Single words are stored in the index.

For more detailed information about text indexing, see the section *Text Retrieval* of the *Advanced Concepts* documentation.

Structure index
A special kind of index is the structure index that **is described above**.

- On the other hand, different technologies can be applied to various index types. From this point of view, we distinguish between the following kinds of index definitions:

Simple indices
These indices represent the normal type of index. They are quite comparable to the indices that may be defined in standard database systems. Such indices have been available in Tamino versions both prior to and following version 4.2.1. The term "simple" in this context denotes that the index is less complex than the other types discussed here.

Multi-path indices
This type of index was introduced in Tamino version 4.2.1.

For more information, see the section ***Multi-Path Indices***.

Compound indices
This type of index was introduced in Tamino version 4.2.1.

Compound indices may only be defined as standard indices.

For more information, see the section ***Compound Indices***.

Reference indices
This type of index was introduced in Tamino version 4.2.1. It is based on the idea that only a sub-tree of the document tree is indexed in this particular index.

For more information, see the section ***The Reference Index: Indexing with Respect to Sub-Trees of a Document***.

Note: For non-XML data, it is also possible to define text indices. This option is described in the section ***Built-In Indexing of Non-XML Data***.

For more information about achieving the best performance when using these advanced indices, see the appropriate section of the *Performance Guide*.

How to Define an Index in Tamino

This can be accomplished either with help from a tool or manually. These options are described in the following sections:

- Defining an Index Using Tamino Schema Editor
- Defining an Index Manually

Defining an Index Using Tamino Schema Editor

Using the Tamino Schema Editor documentation, it is easy to define an index:

To define an index using the Tamino Schema Editor

1 Choose the element or attribute that is to be indexed in the tree-display on the left side of the Tamino Schema Editor.

On the right side of the Tamino Schema Editor, two areas labeled with:

- Logical properties
- Physical properties

will appear.

2 In the **Physical properties** area, set the value for the property `<index>` to "standard" if you intend to define a standard index, or to "text" if you intend to define a text index.

Instances of the chosen element or attribute will be indexed to optimize queries containing relations like comparisons in retrieval expressions.

Note: There is also a "standard+text" option available if you want to define both standard and text indices synchronously.

For more information, see the documentation of the Tamino Schema Editor.

Defining an Index Manually

To define an index manually

- To define a simple index for an element or attribute manually, find the definition of the element or attribute in the schema file and insert a `tsd:index` element as a child element of the corresponding `tsd:native` element in the definition of the element or attribute.

To define advanced types of indices manually, add child elements to `tsd:index` that contain the appropriate information.

Simple text and standard Indices

Simple Indexing

Index Definition Using the `tsd:index` Element

The `tsd:index` element may contain:

A `tsd:text` element
This indicates that the node is indexed for full text retrieval.

A `tsd:standard` element
A standard index is built. This option is usually applied for **data typing**.

Neither
No indexing is performed.

A text index and a standard index may also be defined synchronously. In a simple index, neither the only `tsd:text` element nor the only `tsd:standard` element may have any child element.

XML Indexing Considerations

The challenge in indexing XML data lies in deciding which XML data you wish to store and what the important queries against the stored data are likely to be. Since the "meat" of the data often lies in terminal nodes or sub-trees, the guiding principle is therefore to define an access path, i.e. an index, to those terminal nodes that are likely to be queried. Subtrees that are not relevant for filter conditions do not need to be indexed.

Tamino provides default settings for attributes that allow you to generate a Tamino schema without any editing. By default, neither elements nor attributes are indexed. Thus, for a natively-stored node without index nothing at all needs to be specified. Furthermore, it is very simple to add indexing to a natively-stored node:

To create an index for a natively-stored node

- Specifying an index element with a text child element (i.e. an `tsd:index` element with a `tsd:text` child element) on the root node causes full text indexing in the whole instance tree. Doing this again on individual terminal nodes causes double indexing; this may be useful in cases where you wish to make information accessible via full text searches across a whole doctype as well as via specific search expressions (for example, "Give me patient records that contain the string Atkins", as well as "Give me the record of the patient whose surname is Atkins".

A special case of defining schemas for XML data is the representation of recursive structures in a schema.

The practical management of indexing is explained in the section *Examples*.

The Reference Index: Indexing with Respect to Sub-Trees of a Document

The General Concept of the Reference Index

A simple index indicates that a node belongs to a certain document, but it does not contain any more precise information. This may lead to poor performance if the document contains similar or equal sub-trees occurring with multiplicity that would have to be searched. Tamino provides reference indexing so that you can tune your schema for better performance in these situations.

Reference indices provide improved support for very large documents and for high-complexity documents.

We speak of high complexity if sub-trees with multiplicity occur in the schema. The complexity is even greater if recursive structures occur in the schema.

The general idea behind reference definitions and reference index definitions is to keep an additional index containing only references for a predefined partial tree of the document and to identify these references for later use.

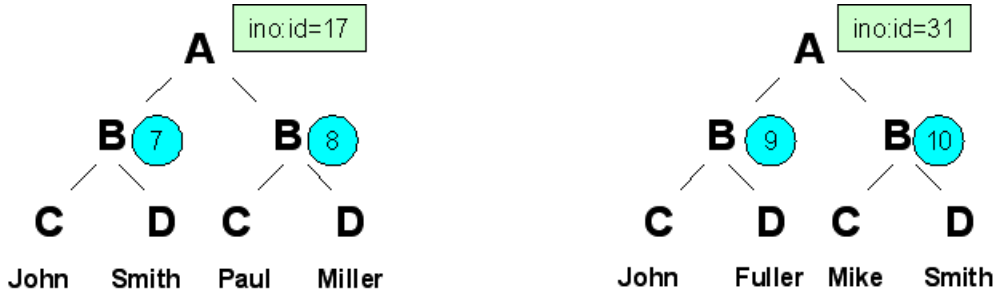
For example, if `/Doc/A/B` is specified as the reference node for the reference index within the contents of the `tsd:refers` element, `B` nodes will be registered in the reference index.

In more detail, this leads to the following consequences:

- In contrast to earlier versions of Tamino, document fragments (partial sub-trees of the document tree) can now be specifically addressed.
- The target node of a reference index is a "dedicated" node. Such nodes define a reference chain.
- Special queries can be significantly accelerated by defining a reference index. A particularly relevant acceleration happens to queries that contain a logical AND operation, if the AND operation is performed relative to a reference node.
- Appropriate index definition is even possible in complex schemas containing recursive definitions.

Example of a Reference Index

Again, assume a document with the structure displayed in the picture below: a root element `A` has multiple children `B`, each having two children named `C` and `D`. Also assume there are two instances with the `ino:id` values 17 and 31 containing the names given in the lowest row of the picture.



We now introduce reference numbers to identify the `B` nodes that appear with multiplicity; we also replace the ino:ids in the right column of the traditional index look-up table by the reference numbers of the `B` nodes that appear with multiplicity.

The look-up table for an index for `A/B/C` based on these instances looks like this:

John	7,9
Mike	10
Paul	8

Reference Index for A/B/C:

Similarly, the look-up table for a reference index for `A/B/D` looks like the following:

Fuller	9
Miller	8
Smith	7,10

Reference Index for A/B/D:

Conclusion

The `tsd:refers` element of TSD can be used in two different ways, depending on the intention:

- For **Reference Definition**: This is done by using it in the context of either `tsd:standard` or `tsd:text`. See the section ***Reference Definition*** for more information;
- For **Reference Index Definition**: This is done by using it in the context of `tsd:reference`.

Reference Definition

You can define a reference based on either a standard index or a text index.

To define a standard index or a text index that references a particular path, add a `tsd:refers` element as a child element of the respective `tsd:standard` or `tsd:text` element. The content of this `tsd:refers` element is the path to the node that is to be used as a base for a reference. (This node is marked as a reference node.) It is given there as an absolute path according to the rules of the **W3C**'s **XPath** language.

This means that not the whole document tree is covered by the index, but only the sub-tree below that path (in the example below it is `/A/B`, for example).

```
...
<xs:element name = "C" type = "xs:string">
...
  <tsd:elementInfo>
    <tsd:physical>
      <tsd:which>
        /Doc/A/B/C
      </tsd:which>
      <tsd:native>
        <tsd:index>
          <tsd:text>
            <tsd:refers>
              /Doc/A/B
            <tsd:refers>
          </tsd:text>
        </tsd:index>
      </tsd:native>
    </tsd:physical>
  </tsd:elementInfo>
...
</xs:element>
...
```

The following rules apply:

- The element describing the index (in the contents of the accompanying `tsd:which` element) must have an ancestor that matches the **XPath** expression of the `tsd:refers` value.
- Only absolute path expressions without wildcards are supported in this context.

As long as these conditions are fulfilled, recursive structures can be indexed.

Reference Index Definition

A node `B` to be referenced in indexing can be defined as follows:

```
...
<xs:element name = "B" maxOccurs = "unbounded">
  ...
  <tsd:elementInfo>
    <tsd:physical>
      <tsd:native>
        <tsd:index>
          <tsd:reference>
            <tsd:refers/>
          </tsd:reference>
        </tsd:index>
      </tsd:native>
    </tsd:physical>
  </tsd:elementInfo>
  ...
</xs:element>
```

If this node has a child element `C` that should have a reference index relative to `B`, this is accomplished as follows:

For more complex schemas and their corresponding documents, TSD's reference index definitions offer the possibility of two-stage modeling, e.g. `/Doc/A/B` refers to `/Doc/A` and `/Doc/A` refers to the document id.

For example, imagine an element named `B` that should have an ancestor named `A` and should be described by the following element definition schema fragment:

The element `B` is defined by this element definition from the schema fragment with a reference index definition that works in such a way that references pointing to node `B` should now point to its parent `/Doc/A`, for which a reference must have been defined.

Note: This schema fragment sets up a reference chain from `B` over `A` to the document root. This reference chain can also be denoted as `B->A->Doc`.

Another motivation for using reference index definitions may come from the fact that recursive structures can also be indexed if the rules given below are not violated.

The following rules apply generally for reference index definitions:

- If a `tsd:which` element has been specified in this context:

  The parent element of the element mentioned within the contents of the accompanying `tsd:which` must match the **XPath** expression specified below the `tsd:refers` element.
- Only absolute path expressions without wildcards are allowed in the **XPath** expression.

- A `tsd:text` or `tsd:standard` element may have only one single `tsd:refers` as a child element.

  Note: However, if you want to define more than one reference index on the same node, you can achieve this by using multiple `tsd:text` or `tsd:standard` elements.

- `tsd:reference` may not be specified for attribute nodes.

The `tsd:refers` child element of the `tsd:reference` element behaves the same as it does in a reference definition as described **above**.

Reference index definitions can also be combined with other index definitions, for example **multi-path indices** and **compound indices**.

Constraints on Reference Indices

- Multiple reference index definitions are not allowed on the same node.

```
<xs:element name = "B" maxOccurs = "unbounded">
  ...
  <tsd:elementInfo>
    <tsd:physical>
      <tsd:native>
        <tsd:index>
          <tsd:reference>
            <tsd:refers>/Doc/A<tsd:refers>
          </tsd:reference>
          ...
        </tsd:index>
      </tsd:native>
    </tsd:physical>
  </tsd:elementInfo>
  ...
</xs:element>
```

The above fragment is valid.

```
<xs:element name = "B" maxOccurs = "unbounded">
  ...
  <tsd:elementInfo>
    <tsd:physical>
      <tsd:native>
        <tsd:index>
          <tsd:reference>
            <tsd:refers>/Doc/A<tsd:refers>
          </tsd:reference>
          <tsd:reference>
            <tsd:refers>/Doc/C/D<tsd:refers>
```

```
          </tsd:reference>
          ...
        </tsd:index>
      </tsd:native>
    </tsd:physical>
  </tsd:elementInfo>
  ...
</xs:element>
```

The above fragment is invalid: multiple `tsd:reference` elements are not allowed.

- Defining a reference index is only allowed if the structure index is set to either "condensed" or "full".

You can find a more detailed example below in the section *Example 6: Defining a Reference Index*. For example, the index definitions made there accelerate the processing of the following queries:

- `/Doc/A/B[C~='my' and D=1]`
- `/Doc/A[B[C~='my' and D<1]]`
- `/Doc[A/B[C~='my' and D<1]]`

However, the following queries do not benefit from the definition of a reference index:

- `/Doc/A[B/C~='my' and B/D<1]]`

  Note: Indeed, for this query the execution time is longer if a reference index is defined on `B` (compared to a simple index).

- `/Doc[A/B/C~='my' and A/B/D>2] index lookup C and D`

  Note: This query is more efficient with a simple index than with a reference index.

General Rules for Index Combination

The following rules concerning combination possibilities apply to reference indices:

- A reference index can be defined along with a standard or a text index.
- Reference indices can be combined with both compound indices and multi-path indices.
- Within one `tsd:index` element, currently only one `tsd:text` element containing a `tsd:refers` element is allowed.

Advantages and Disadvantages of Reference Indexing

Advantages

- A reference index can be used in combination with any other kind of index (standard, text, compound, multipath).
- Improved selectivity by combining values relative to subtrees.

Disadvantages

- Sorting over the index with `tsd:refers` is not possible.
- The number of documents in a doctype is limited by the number of allowed entries.

Effect of Reference Index Definitions on Tamino Limits

The number of documents that can be stored in a Tamino database decreases significantly when reference indices are used, especially when multiple definitions of a reference index have been made in this doctype.

Effects on Performance

The correct choice of reference indices can have a significant effect on performance. This is discussed in more detail in the *Performance Guide*.

Multi-Path Indices

In contrast to previous versions of Tamino, where every index reflected an absolute **XPath** address, it is now possible to define an index in Tamino whose **XPath** address matches special criteria. This index is called a multipath index. The definition of multi-path indices is discussed here under the following headings:

Motivation

The reasons that lead to the introduction of multi-path indices are as follows:

- Versions of Tamino prior to 4.2.1 did not support indices for recursive or highly nested structures. In order to improve the usability of Tamino in such situations and also to avoid possible loss of performance, as from Tamino 4.2.1 both recursive and highly nested structures can be indexed using multi-path indices.
- In earlier Tamino versions, index support was missing for queries that contain wildcard expressions, for example `/Play//Title`. Queries containing wildcard expressions performed badly, due to the time-consuming analysis of all matching indices with respect to the given predicate.

The General Concept of Multi-Path Indexing

A multi-path index allows data from multiple paths to be collected in a single index.

Whereas in simple indexing each index reflects an absolute **XPath** address, this does not hold for a multi-path index.

A multi-path index is filled by the data of all nodes that match the various **XPath** conditions. This increases the performance for queries with wildcards significantly, as only the sum of all relevant entries is searched (and not all possible combinations, as in a simple index).

A multi-path index can be defined either as a text index or as a standard index, depending on which was specified as the parent element of the `tsd:multiPath` element in the schema.

It is also possible to combine multi-path indexing with any other kind of indexing within one node:

- Simple standard index;
- Simple text index;
- Compound index.

Practically, a multi-path index is defined as follows:

All nodes (this includes both element nodes and attribute nodes) that should contribute to the multi-path index are identified by a special label that is unique within the entire schema. This allows you to specify easily the values that should be assembled into each index.

In the schema definition for a multi-path index, this is simply expressed as an additional element `tsd:multiPath` which contains the common label to be used in the index definitions of all nodes that should be indexed. All nodes (i.e. elements and attributes) that have the same label in their index definition address the same index.

For example, as you can find in the contents of the `tsd:multiPath` element, the three labels `MultiPathIndex0`, `MultiPathIndex2` and `MultiPathIndex3` are used in the example below.

This approach offers the following solutions:

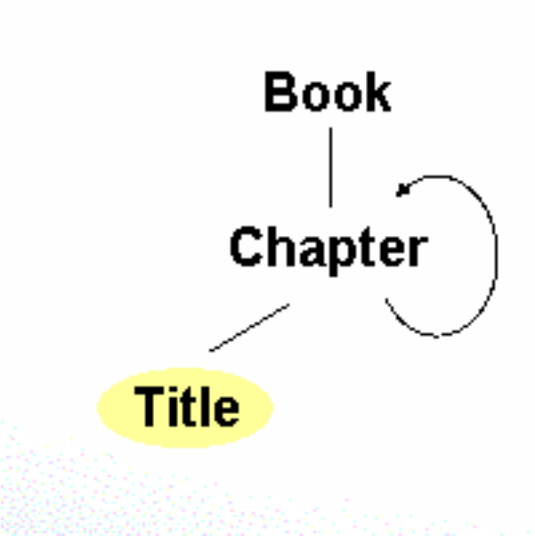
- It is now possible to define an index on nodes that are part of a recursive schema. Multi-path indexing provides efficient indexing, even on recursive structures that could not be indexed in earlier versions of Tamino.
- Support for highly connected schemas is offered.

  Highly connected schemas in this sense are schemas that contain nodes that are referenced many times from other nodes (for example by usage of the `xs:any` or `xs:anyAttribute` elements within the schema).

Example Schema Excerpts for Multi-Path Indexing

First we discuss a simple example, showing the use of multi-path indexing in a recursive situation.

Imagine, for example, an element `title` that may be addressed as a child element within a recursive schema. See the illustration below:



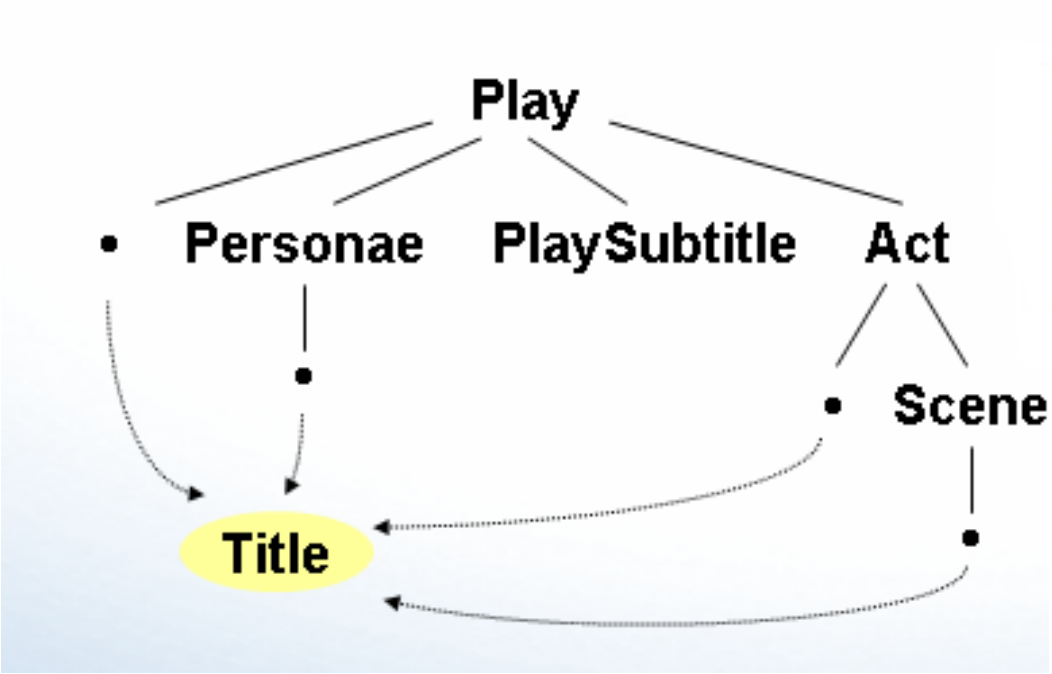
To create a multi-path index for the element `title`, add the following lines to its `tsd:native` element:

```
.
.
.
<tsd:index>
  <tsd:text>
    <tsd:multiPath>Title index<tsd:multiPath>
  </tsd:text>
</tsd:index>
.
.
.
```

This code creates a multi-path index named Title index that covers all possible access paths that lead to `title` by recursion. One single index covers all recursion levels.

Now let us have a look at a simple example for highly connected schemas.

Imagine, for example, an element `title` that can be addressed via multiple paths within a highly connected schema. See the illustration below:



To create a multi-path index for `title`, simply add the same lines as in the preceding example to its `tsd:native` element:

```
.
.
.
<tsd:index>
  <tsd:text>
    <tsd:multiPath>Title index<tsd:multiPath>
  </tsd:text>
</tsd:index>
.
.
.
```

This code creates a multi-path index named Title index covering all access paths that lead to `title`. However, index generation does not depend on the path leading to `title` in this case.

The generated multi-path index accelerates queries such as:

```
Play [ .//Title ~= "King" ]
```

Finally, let us discuss a more complex example:

```
.
.
.
<!-- global Element Definition for Title -->
  <xs:element name = "Title" type = "xs:string">
    <xs:annotation>
      <xs:appinfo>
        <tsd:elementInfo>
          <tsd:physical>
            <!-- default which for global Element Definition Title -->
            <tsd:native>
              <tsd:index>
                <tsd:text>
                  <tsd:multiPath>MultiPathIndex0</tsd:>
                </tsd:text>
              </tsd:index>
            </tsd:native>
          </tsd:physical>
          .
          .
          .
          <tsd:physical>
            <tsd:which>/Play/Act/Title</tsd:which>
            <tsd:native>
              <tsd:index>
                <tsd:text>
                  <tsd:multiPath>MultiPathIndex2</tsd:multiPath>
```

```
              </tsd:text>
            </tsd:index>
          </tsd:native>
        </tsd:physical>
        <tsd:physical>
          <tsd:which>/Play/Act/Scene/Title</tsd:which>
          <tsd:native>
            <tsd:index>
              <tsd:text>
                <tsd:multiPath>MultiPathIndex3</tsd:multiPath>
              </tsd:text>
            </tsd:index>
          </tsd:native>
        </tsd:physical>
      </tsd:elementInfo>
    </xs:appinfo>
  </xs:annotation>
</xs:element>
.
.
.
<!-- local Element Definition for /Play/Prologue /Title -->
<xs:element name = "Title" type = "xs:string">
  <xs:annotation>
    <xs:appinfo>
      <tsd:elementInfo>
        <tsd:physical>
          <tsd:native>
            <tsd:index>
              <tsd:text>
                <tsd:multiPath>MultiPathIndex2</tsd:multiPath>
              </tsd:text>
            </tsd:index>
          </tsd:native>
        </tsd:physical>
      </tsd:elementInfo>
    </xs:appinfo>
  </xs:annotation>
</xs:element>
.
.
.
<!-- local Element Definition for /Play/Act/Abstract/Title -->
<xs:element name = "Title" type = "xs:string">
  <xs:annotation>
    <xs:appinfo>
      <tsd:elementInfo>
        <tsd:physical>
          <tsd:native>
            <tsd:index>
              <tsd:text>
                <tsd:multiPath>MultiPathIndex3</tsd:multiPath>
```

```
              </tsd:text>
            </tsd:index>
          </tsd:native>
        </tsd:physical>
      </tsd:elementInfo>
    </xs:appinfo>
  </xs:annotation>
</xs:element>
.
.
.
```

This definition creates three multi-path indices that cover all `Title` nodes, namely:

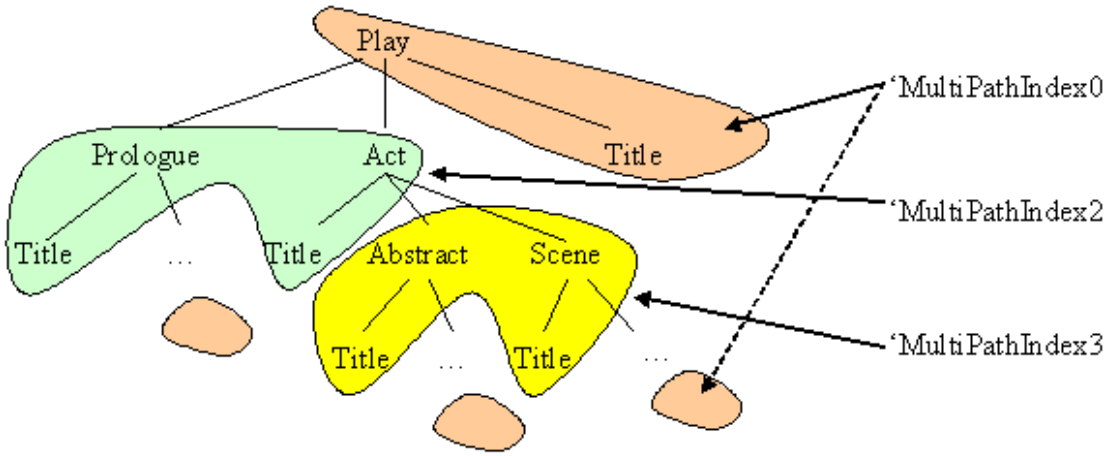
- An index for `/Play/*/Title`, labeled as `MultiPathIndex2`, containing:
  - Data from the global title element restricted by the `<tsd:which>/Play/Act/Title`, and
  - Data from the local element in `/Play/Prologue`.

  This index covers the green area in the illustration.
- An index for `/Play/Act/*/Title`, labeled as `MultiPathIndex3`, containing:
  - Data from the global title element restricted by the `<tsd:which>/Play/Act/Scene/Title`, and
  - Data from the local element in `/Play/Act/Abstract`.

  This index covers the yellow area in the illustration.
- An index for all other `Title` nodes, labeled as `MultiPathIndex0`.

  This index covers the light brown area in the illustration.

This situation is depicted in the following graphic:



You can find another example in the section *Example 4: Defining of a Multipath Index*. For example, the index definitions discussed there accelerate the processing of the following queries:

- ```
  /Doc[A//E/F ='Hy']
  ```
- ```
  /Doc/A[.//E/F ='Hy']
  ```
- ```
  /Doc/A[B/C='my' and .//E/F='Hy']
  ```

General Rules for Index Combination

The following rules concerning combination possibilities apply to multi-path indices:

- A multi-path index may be defined along with a standard or a text index.
- A multi-path index can be combined with both compound indices and reference indices.

Constraints and Limitations

The following constraints apply to the definition of multi-path indices:

- The combination of indices from different paths into one physical multi-path index requires that the participating indices all be of the same kind.

  The following combinations are invalid:

  - Combination of standard and text indices.
  - Combination of different datatypes:

    All datatypes of nodes (i.e. elements and attributes) contributing to the multi-path index must be of the same common datatype.

    For standard indices, a common **XML Schema** datatype must be chosen for all nodes. For compound indices, the datatypes of all parts that are combined must be the same.
  - Combination of different collation specifications:

    Items with non-matching `tsd:collation` elements are rejected. All child elements of `tsd:collation` must match.
  - Combination of compound and non-compound indices.
  - Combination of compound indices with different layouts (number of components and sequence of datatypes).

    Note: However, the paths may differ.
  ```

- The label contained in the `tsd:multiPath` element must not be empty.
- If you want to work with multi-path indices, the **structure index** must have been activated by setting it to either "condensed" or "full".

Advantages and Disadvantages of the Use of Multi-Path Indexing

Advantages

The main advantage of multi-path indexing is easy indexing of recursive and highly-connected structures.

Disadvantages

The main disadvantage of multi-path indexing is that sort operations are not supported.

Compound Indices

This section discusses the following topics related to compound indices:

Motivation

In some queries, combined value conditions on inner nodes (i.e. nodes that are neither the root note nor leaf nodes) appear with a multiplicity of more than one. For example, `/A[B[C="x" and D="y"]]` represents such a query. Characteristic for this construct is:

- the multiplicity of `B` greater than 1, and:
- there is an AND operation below `B`.

In general, such queries perform poorly using simple indices. This is because the simple index delivers a superset of the final result, which must then be filtered in a subsequent processing step. The compound indices that are available in Tamino since version 4.2.1 are ideally suited to accelerate such queries.

Note: Reference indices are also well suited in this situation. To decide whether to use compound indexing or reference indexing, read the sections about the advantages and disadvantages of each kind of indexing in this document.

General Concept

A compound index is somewhat different from the other kinds of index discussed here. A simple index, a reference index and a multi-path index (as discussed above) are each bound to one specific element or attribute. This not true for a compound index, which comprises two or more different components, called fields. The definition, however, takes place in the inner node (`B`), which is the parent of both elements in the condition (`C` and `D` in the example).

To define a compound index, add a `tsd:field` element as a child element directly below the `tsd:standard` element.

The `tsd:field` element must appear at least twice (with different `xpath` attributes) to define a valid compound index.

Note: If the `tsd:field` element appears as a child element of the `tsd:standard` element, in order to define a valid compound index it must appear at least twice, since at least two fields are required to set up a compound index. The `xpath` attributes of at least two `tsd:field` elements belonging to the same `tsd:standard` element must be different.

You can use the `xpath` attribute of the `tsd:field` element to specify a path to an element or attribute contributing to the compound index.

Important: The `xpath` attribute is required.

A `tsd:field` element with a corresponding `xpath` attribute must be defined for each **XPath** address to be included in the compound index. Each path is relative to the element in which it is defined.

Note: The order of the `tsd:field` elements is significant; it determines the order in which the values contribute to the index entry.

Supported `xpath` attributes are:

Sequences of element names (optionally followed by a single attribute name), separated by slashes

Any **XPath** expression that uses only the axes “child” and “attribute”.

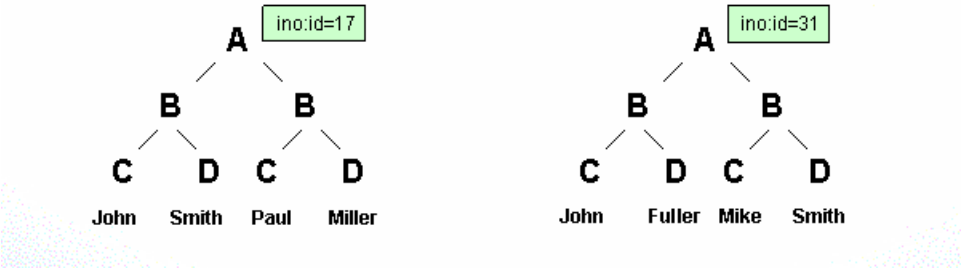
A single dot

The special case of a single dot “.”, which indicates that the index is to be created from the value of the element and of the values of its attributes.

Caution: It is possible, although generally not desirable, to define a schema with multiple equivalent definitions of compound indices. In this case, redundant indices are created.

Example for a Compound Index

Assume a document with the structure displayed below with a root element `A` having multiple children `B`, each having two children named `C` and `D`. Also assume there are two instances with the ino:id values 17 and 31 containing the names given in the lowest row of the picture.



The look-up table for a compound index for `B(C,D)` based on these instances is as follows:

John_Fuller	31
John_Smith	17
Mike_Smith	31
Paul_Miller	17

Compound Index for B(C, D)

Example Schema Excerpt for Compound Indexing

Let `A` be the root element of a document. `A` has child elements `B` and `C` (there may also be others). `B` has an attribute named `b`.

The following example defines a compound index for the combination of attribute `b` of element `B` and the element `C`:

```
<xs:element ...>
  .
  .
  .
  <tsd:elementInfo>
    <tsd:physical>
      <tsd:native>
        <tsd:index>
          <tsd:standard>
            <tsd:field xpath="C"/>
            <tsd:field xpath="B/@b" />
          </tsd:standard>
          .
          .
```

```
          .
        </tsd:index>
      </tsd:native>
    </tsd:physical>
  </tsd:elementInfo>
</xs:element>
```

The compound index created by this schema definition contains an entry for each combination of `C` and `B/@b` occurring below the element in whose `tsd:elementInfo` it appears.

The performance of queries such as `/A[@b="x" and C="y"]` is improved by this compound index.

You can find a more detailed example in the section ***Example 3: Defining a Compound Index*** of this document. The index definitions discussed there improve the performance of the following queries:

- `/Doc/A/B[C= 'my' and D=1]`
- `/Doc/A[B[C= 'my' and D<1]]`
- `/Doc[A/B[C= 'my' and D<1]]`

General Rules for Index Combination

The following rules concerning combination possibilities apply to compound indices:

- A compound index may be defined only as a standard index.
- A compound index can be combined with a reference index or with a multi-path index.
- The order of the components in the definition of the compound index is significant.
- You can define a compound index that covers more than two fields.
- You can define an arbitrary number of compound indices for one schema, as long as the limit on the total number of indices required by a schema is not violated.
- Components with arbitrary datatypes and collations are supported.

Constraints and Limitations

The following constraints apply for the definition of compound indices:

- A compound index definition with exactly one `tsd:field` element is invalid.
- If the path specified in the `xpath` attribute of one or more `tsd:field` child elements of the `tsd:standard` element is invalid in the sense of the ***W3C XML Path Language (XPath) Version 1.0 specification***, then the compound index specification is invalid in its entirety.
- It is not possible to define a compound index definition for text indices, see **above**.

- A compound index whose components point to nodes with mapping to Adabas, SQL or Tamino Server Extensions is not allowed.
- A compound index may not be defined below attributes. It can only be defined on an element.

Advantages and Disadvantages of Compound Indexing

Advantages

- A compound index can be used together with all kinds of indices except text indices (standard, reference and multipath indices are supported).
- Improved selectivity by combining values relative to subtrees.
- Only one index look-up instead of multiple index look-ups.
- Sort over all components is possible.

Disadvantages

- Compound index definitions are not available for text indexing.

Other Sources of Information on Indexing

- The *Tamino XML Schema Reference Guide*. Includes the descriptions of all TSD language elements that are required for index definition in Tamino schemas.
- The *Tamino Performance Guide*.
- The *Advanced Concepts* documentation. It also discusses indexing on a more general level.
- The documentation on *X-Machine Programming*. The ino:DisplayIndex function that can display index data is described there.

External Mapping

To map parts of XML documents to externally stored data (Adabas, or using a Tamino Server Extension), use the `tsd:map` element.

An example showing a mapping for an element follows:

```
<xs:element>
 <xs:annotation>
  <xs:appinfo>
   <tsd:elementInfo>
    <tsd:physical>
     <tsd:map>
     .
```

```
      .
      .
     </tsd:map>
    </tsd:physical>
   </tsd:elementInfo>
  </xs:appinfo>
 </xs:annotation>
 .
 .
 .
</xs:element>
```

The `tsd:map` element may contain an optional `tsd:ignoreUpdate` child element. If present, Tamino does not pass the corresponding part of the XML document to the internal data storage or X-Node during processing.

This section covers the following topics:

- *Mapping to Adabas Files*
- *Mapping to Server Extensions*

Mapping to Adabas Files

The Tamino schema provides constructs that allow you to store data in an **Adabas** file and/or retrieve data from an Adabas file via the Tamino X-Node.

Mapping to **Adabas** is done on a file basis. In other words, you do not have to model the whole of an **Adabas** database using the Tamino schema language; you model only those files and fields that you wish to access using Tamino.

Mapping to an **Adabas** file is explained under the following headings:

- *Elements and Attributes for Adabas Mapping*
- *Example Schema for Adabas Mapping*

Pure vs. Hybrid Adabas Mapping

There are two possible approaches for mapping to an Adabas file:

- **Pure Mapping**
  This means storing the data for a document in a single Adabas file.

  Pure XA Mapping
  Pure XA mapping defines a mapping from any XML schema to Adabas. All XML data can be stored in an Adabas file and accessed via normal XML operations, using Adabas as the XML store.

  This option is especially suited for mapping from existing XML schemas to Adabas files.

Pure AX Mapping

Pure AX mapping defines a mapping from any Adabas schema to an XML schema. Correspondingly, all XML operations are mapped to Adabas operations. All Adabas data can be accessed via XML operations (XML view on Adabas data).

This option is especially suited for mapping from existing Adabas files to XML schemas.

A pure mapping is denoted by the presence of a `tsd:pure` child element below the `tsd:map` element; otherwise, the mapping is hybrid. The `tsd:pure` element is described in the next subsection and in the section *`tsd:pure` element* of the *Tamino XML Schema Reference Guide* .

In older versions of Tamino, the X-Node implementation was only capable of representing a hybrid XA mapping. The `tsd:pure` element now enables pure XA mapping. Pure Adabas mapping is specified as a physical option for a doctype:

```
<tsd:doctype ↩
xmlns:tsd="http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
 <tsd:physical>
  <tsd:map>
   <tsd:pure/>
  </tsd:map>
 </tsd:physical>
</tsd:doctype>
```

- **Hybrid Mapping**

  Hybrid Mapping means storing the data for a document partly in Tamino and partly in a single Adabas file. This mapping option has been supported by all former versions of Tamino.

  Hybrid XA Mapping

  Hybrid XA mapping represents the combination of several mappings together with native storage parts into one document via a single schema. Varying degrees of hybrid mapping are possible.

  Hybrid AX Mapping

  Hybrid AX mapping integrating existing Adabas and natively-stored data into a single XML view (XML view on Adabas and native XML data).

Data is handled in the following manner if a pure Adabas mapping is present:

-  **Important:** In such a doctype only one Adabas file can be mapped.

- No data is stored in Tamino for that doctype.

  Any real content in the document must be mapped to Adabas.

- The only permitted manipulation operations on pure Adabas mapped doctypes are:

`_process`
To insert new documents.

In particular, inserting new documents does not assign an `ino:id`, and any markup (for example comments, processing instructions or insignificant whitespace) is lost.

XQuery Update
XQuery Update is the only possible means to update and delete documents.

Even though a document was inserted through Tamino, it cannot be updated with `_process`, and it cannot be deleted with `_delete` (updating and deletion are only possible through `_xquery` update).

`_xquery` update does not store anything in Tamino, any modifications of the markup (e.g. comments) are lost.

- You cannot use a full structure index in conjunction with pure mapping to Adabas.

The following conditions must be fulfilled in order to establish a pure Adabas mapping:

- The root element referenced by the doctype mapped to pure must contain one Adabas file mapping (`tsd:subTreeAdabas`).
- Each element declaration of simple content and each attribute declaration must contain one Adabas field mapping (`tsd:nodeAdabasField`).
- An element declaration of complex content is not allowed to contain mixed content (`xs:complexType mixed="false"`).
- The doctype must not contain recursions.
- The doctype does not allow for open content, i.e. `<tsd:content>open</tsd:content>` is not allowed.
- Elements with `type="xs:anyType"` are not allowed.
- No element without `xs:simpleType`, `xs:complexType`, or type attribute may occur.
- No wildcard (`xs:any`, `xs:anyAttribute`) with `processContents = "skip"` or `"lax"` is allowed.
- The root node must be mapped to an Adabas file.
- Each descendant of the root node must meet one of the following conditions:
  - It has a complex content model that does not allow for mixed content and is not empty;
  - It has simple content that is mapped to an Adabas field.
- The same Adabas field may not be mapped to different paths. Otherwise a run-time error might be signaled.

Plain URL Addressing

Normally, when a document is inserted using plain URL addressing (HTTP PUT), Tamino returns the `ino:id` of the new document in the HTTP header field `X-INO-ID`. For pure X-Node Adabas

doctypes, there is no `ino:id` when inserting a document, and therefore the `X-INO-ID` field contains zero. Consequently, the pure X-Node Adabas documents cannot be read using plain URL addressing (HTTP GET), even if they were inserted through Tamino.

Elements and Attributes for Adabas Mapping

This section describes the attributes that are of particular relevance to **Adabas** mapping.

Adabas-Specific Elements and Attributes

`tsd:subTreeAdabas` **element**

This element provides the database ID and file number in its attributes `dbid` and `fnr`, with an optional password in the `password` attribute.

`dbid` **attribute**

The `dbid` attribute belongs to the `tsd:subTreeAdabas` element. Its value contains the database ID of the Adabas database.

`fnr` **attribute**

The `fnr` attribute belongs to the `tsd:subTreeAdabas` element. It represents the file number of the Adabas file.

`password` **attribute**

The optional `password` attribute belongs to the `tsd:subTreeAdabas` element. It contains the password of the Adabas file to be accessed. The password has up to 8 characters.

`tsd:subTreeAdabasPE` **element**

This element models a mapping to an **Adabas** periodic group via Tamino X-Node.

`tsd:nodeAdabasField` **element**

This element is used in conjunction with the elements `tsd:subTreeAdabas` and `tsd:subTreeAdabasPE` to specify the characteristics of the leaf nodes.

`shortname` **attribute**

The `shortname` attribute can be an attribute either of the `tsd:subTreeAdabasPE` element or of the `tsd:nodeAdabasField` element. This is the shortname (e.g. “AH”) of the corresponding field in the **Adabas** file. It can also be used in conjunction with Adabas MU fields.

Note: This is the case if a `tsd:multiple` child element is present inside `tsd:nodeAdabasField`.

`format` **attribute**

The `format` attribute belongs to the `tsd:nodeAdabasField` element. It describes the format of the corresponding field in the Adabas file.

The following table presents a list of the possible `format` attribute values and their meaning:

format	Description/Meaning
A	Alphanumeric
B	Binary
F	Fixed Point
G	Floating Point
P	Packed Decimal
U	Unpacked Decimal

Also refer to the Adabas documentation.

Note: The `format` attribute can be specified either for a `tsd:nodeAdabasField` element describing a mapping to a single Adabas field (that does not have a child element `tsd:multiple`) or for those describing a mapping to an Adabas MU field (multiple field) `tsd:nodeAdabasField` (that can have a child element `tsd:multiple`).

Example Schema for Adabas Mapping

This example uses the *Employees* file in the demo database delivered with the Windows installation of Adabas Version 3.2 to illustrate mapping to an Adabas file.

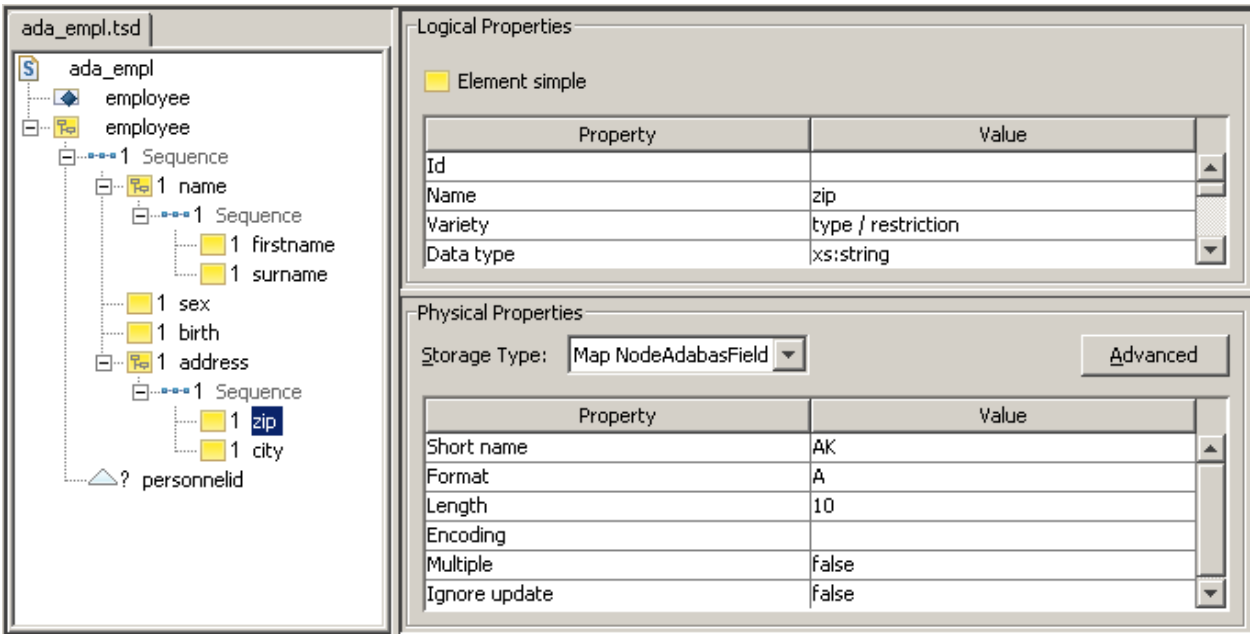
The following is the Natural view of the "Employees" database:

Note: The lines of special interest are printed in *italics*.

```
T L DB Name                            F Leng  S D Remark
- - -- --------------------------------  - ----  - - ------------------------
  1 AA PERSONNEL-ID                     A    8    D
       HD=PERSONNEL/ID
G 1 AB FULL-NAME
  2 AC FIRST-NAME                       A   20  N
  2 AD MIDDLE-I                         A    1  N
  2 AD MIDDLE-NAME                      A   20  N
  2 AE NAME                             A   20    D
  1 AF MAR-STAT                         A    1  F
       HD=MARITAL/STATUS
  1 AG SEX                              A    1  F
       HD=S/E/X
  1 AH BIRTH                            N  6.0    D
       HD=DATE/OF/BIRTH
       EM=99/99/99
G 1 A1 FULL-ADDRESS
M 2 AI ADDRESS-LINE                     A   20  N
       HD=ADDRESS
*      OCCURRENCES 1-6
  2 AJ CITY                             A   20  N D
  2 AK ZIP                              A   10  N
```

```
        HD=POSTAL/ADDRESS
  2 AK  POST-CODE                       A   10  N
        HD=POSTAL/ADDRESS
  2 AL  COUNTRY                         A    3  N
G 1 A2  TELEPHONE
  2 AN  AREA-CODE                       A    6  N
        HD=AREA/CODE
  2 AM  PHONE                           A   15  N
        HD=TELEPHONE
.
.
.
```

The file *ada_empl.tsd* is provided in the Tamino distribution kit to illustrate the mapping to the *Employee* database. Note that not all fields are mapped, so instances containing nodes that are not mapped are rejected. The graphical representation of this schema as it appears in the Tamino Schema Editor is shown below. The properties view is reduced to the physical properties pane, which shows as an example the mapping of the `zip` element to the Adabas field `AK`:



Note that such a schema could also be a subtree within a schema for another doctype.

This schema allows Adabas data to be retrieved using Tamino X-Query expressions, for example:

```
/employees/employee[name/surname~="A*"]
```

This returns all employees whose surnames start with the letter “A”.

The following XML object is an instance of the above schema and can be loaded into the Adabas database and retrieved using Tamino:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <employee personnelid="007">
  <name>
   <firstname>James</firstname>
   <surname>Bond</surname>
  </name>
  <sex>M</sex>
  <birth>340401</birth>
  <address>
   <zip>SW13JB</zip>
   <city>London</city>
  </address>
</employee>
```

Mapping to Server Extensions

Server Extensions allow you to extend Tamino's functionality by executing user-defined application logic (written in languages such as Java or C++) at specified points in the processing of an instance. Server Extensions can be associated with elements and attributes at any level in a Tamino schema. When an element or attribute is associated with a Server Extension, the element is said to be “mapped” to the Server Extension.

Server Extensions can be used, for example, to let external programs handle indexing, storing or retrieval of data in ways not provided as standard by Tamino.

Mapping to a Server Extension means transferring control to an external function offered by the Server Extension. One Server Extension may offer several functions. Control can be passed to a Server Extension function during parsing of an incoming instance or during composition of an XML object as the result of a query. The invocation context is specified by the name of the Server Extension-specific child elements on the node. The associated Server Extension function is specified as the element value.

Mapping to Server Extension functions is described in this section under the following headings:

- **Node Attributes for Mapping to Server Extensions**
- **Example of Mapping to Server Extensions**

Node Attributes for Mapping to Server Extensions

This section describes the Tamino schema extensions that are of particular relevance to mapping elements to Server Extensions.

Server Extension-Specific Elements and Attributes

There is one single parent element in TSD offering all the functionality needed to map Tamino Server Extensions in conjunction with its child elements, namely the `tsd:xTension` element.

TSD provides the following mappings for Server Extensions:

`tsd:xTension`
: This element is the parent element of the elements `tsd:onDelete` and `tsd:onProcess`.

  The syntax of the values of these child elements is:

```
SXSModule.SXSFunction
```

`tsd:onProcess`
: Specifies the name of the SXS function that is to be executed when data is stored using Tamino (that is, when using the Tamino `_process` request). The SXS function is called *after* parsing the associated element.

`tsd:onDelete`
: Specifies the name of the SXS function that is to be executed when the associated element is to be deleted (that is, when the Tamino `_delete` request is issued).

Example of Mapping to Server Extensions

The following example demonstrates the mapping to a Server Extension Function. A doctype "Mail" describes the structure of an e-mail message.

The effect of the Server Extension function is that instead of storing instances of the doctype (i.e., e-mail messages) in Tamino, they are routed to an e-mail tool, which transmits the messages to the recipient.

The TSD schema describing the "Mail" doctype with mapping to Server Extensions contains a `tsd:onProcess` element specifying the corresponding Server Extension function to be invoked when processing a document:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema"
 xmlns:tsd = "http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
  <xs:annotation>
    <xs:appinfo>
      <tsd:schemaInfo name = "SXS">
        <tsd:collection>MailSXS</tsd:collection>
        <tsd:doctype name = "Mail">
          <tsd:logical>
            <tsd:content>closed</tsd:content>
            <tsd:accessOptions>
              <tsd:insert></tsd:insert>
            </tsd:accessOptions>
          </tsd:logical>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name = "Mail">
    <xs:annotation>
      <xs:appinfo>
        <tsd:elementInfo>

        <!--- Beginning of mapping to Server Extension--->
          <tsd:physical>
            <tsd:map>
              <tsd:xTension>
                <tsd:onProcess>SXSMail.SendMail</tsd:onProcess>
              </tsd:xTension>
            </tsd:map>
          </tsd:physical>
        <!--- End of mapping to Server Extension--->
        </tsd:elementInfo>
      </xs:appinfo>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "Recipient" type = "xs:string"></xs:element>
        <xs:element name = "Subject" type = "xs:string"></xs:element>
        <xs:element name = "Body" type = "xs:string"></xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Notes:

1. In this example, the Server Extension function is specified on the root node of the doctype, thus ensuring that the function is executed after parsing the whole instance. It could also be placed at a subordinate node of a document node.

2. Mapping to a Server Extension function means that the corresponding subtree of the XML document is not stored in Tamino, but is processed by the map-in function (see *Tamino Server Extension Functions*) being invoked.

The following XML object is an instance of the doctype “Mail”. Using the schema definition above, this instance is not stored in Tamino, but sent to the e-mail address in the `Recipient` element:

```
<Mail>
  <Recipient>world@planet.org</Recipient>
  <Subject>Global greeting</Subject>
  <Body>Hello World!</Body>
</Mail>
```

7 Schema-Related Attributes in XML Documents

The following topics are discussed in this chapter:

xsi:type

Tamino fully supports the `xsi:type` attribute. The `xsi:type` attribute indicates that within an instance a type other than the element's default type as specified in the schema is to be used for validation.

Assume the following schema:

```
<xs:element name="bill" type="bill"/>
<xs:complexType name="bill" abstract="true">
  <xs:simpleContent>
    <xs:extension base="xs:integer"/>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="USDollar">
  <xs:simpleContent>
    <xs:restriction base="bill">
      <xs:enumeration value="1"/>
      <xs:enumeration value="5"/>
      <xs:enumeration value="10"/>
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="Euro">
  <xs:simpleContent>
    <xs:restriction base="bill">
      <xs:enumeration value="5"/>
      <xs:enumeration value="10"/>
      <xs:enumeration value="20"/>
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="Rupee">
  <xs:simpleContent>
    <xs:restriction base="bill">
      <xs:enumeration value="100"/>
      <xs:enumeration value="200"/>
      <xs:enumeration value="500"/>
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
```

Then the following documents are valid:

```
<bill xsi:type="USDollar" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  10
</bill>
```

```
<bill xsi:type="Rupee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  100
</bill>
```

but this document is invalid:

```
<bill xsi:type="Euro" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  25
</bill>
```

Usage of the `xsi:type` attribute is similar to the usage of substitution groups: the referenced type must be derived from the default type as of the schema. In our example, the default type `bill` is abstract. Hence it must not be used in an XML document and usage of `xsi:type` is even enforced.

`xsi:type` was partially supported in Tamino version 4.2, i.e. it validated against the referenced type, but did not check the type hierarchy. Usage of `xsi:type` was only allowed if the server parameter `reject xsi:type` was set to "no". This parameter has now been dropped, and `xsi:type` is fully supported.

xsi:nil

Nillable elements in Tamino correspond to NULL values in SQL. The following example illustrates how an element may be declared in the schema as nillable:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
      <xs:element name="dateOfBirth" type="xs:date" nillable="true"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The following document is valid against the schema:

```
<person>                                                                    ↩

  <firstname>Big</firstname>                                                ↩

  <lastname>Unknown</lastname>                                              ↩

  <dateOfBirth xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
</person>                                                                   ↩
```

It indicates that the value of the `<dateOfBirth>` element is unknown.

In addition to supporting bare validation as described by the **XML Schema** recommendation, Tamino supports nillable elements in XNode mapping for Adabas and SQL:

- When mapping an element with `nillable="true"` to an Adabas field or SQL column, Tamino checks that the NC option is set (Adabas) or that the column does not have the NOT NULL property (SQL);
- `xsi:nil="true"` is mapped to the DBMS-specific NULL-indicator and vice versa.

xsi:schemaLocation

This attribute is ignored by Tamino, since Tamino has its own mechanism for determining the schema document to be used for validation.

xsi:noNamespaceSchemaLocation

This attribute is ignored by Tamino.

8 Schema Operations

This chapter discusses the following topics:

Defining a Schema

The operation of defining a schema can be divided into two categories:

- Simple Schema Definition
- Defining a Cluster of Schemas

Simple Schema Definition

The operation of simple schema definition is explained in the section *The Logical Schema*. The steps described there apply to simple schema definition.

Defining a Cluster of Schemas

It is possible to define multiple schemas, i.e. a cluster of schemas, in a single request. This makes it possible:

- To define a set of schemas that have circular dependencies without having to manually break the cycle;
- To define a set of related schemas in any order.

It is also possible to delete multiple collections, schemas or doctypes with a single **_UNDEFINE** command.

These features are fully supported by the Tamino Schema Editor.

Updating Existing Schemas (Update Schema)

The X-Machine `_define` command can be used to update existing schemas. This section describes some aspects to be considered when updating schemas, and lists the restrictions involved.

This information is organized under the following headings:

- General Considerations
- Changing Element and Attribute Values in the Physical Schema
- Schema Evolution for Open Content (Update Schema Processing)
- Update Schema Checks for Imported Schemas

- Update Schema Checks for xsi:type

General Considerations

Important: The guiding principle to updating existing schemas is the following: It is guaranteed that all documents already stored in a doctype also remain valid with respect to the updated schema and the doctypes defined therein. If the schema is too strict, it must be loosened.

It is very difficult to detect all cases where a schema modification still permits all instances to be valid with respect to the new schema, for example if an element's content model is changed from arbitrary content (e.g. if untyped) to an explicit content model using a complex type definition. In this case, instances that are already loaded into Tamino may or may not validate against the new schema. When defining the new schema, it is possible to pass a parameter `_mode=validate` with the `_define` command. This causes Tamino to first perform a structure-based comparison of the old schema and the new schema. If Tamino cannot guarantee that all instances validate against the new schema, it then explicitly validates all instances against the new schema. If no errors are detected, the new schema is accepted. This can be done using the Tamino Schema Editor.

Note: Adding indices and adding or changing collations are potentially time-consuming operations, due to the doctype scan. This is also true for `_define` with `_mode=validate`.

This implies:

Logical Schema

- Structural updates are restricted to adding optional nodes (elements or attributes).

  When adding new nodes to a schema, the same restrictions as for using the `_define` statement generally apply (see *Mapping Type Dependencies*).
- Type changes are not possible, except that existing enumerations can be extended and constraining facets such as `xs:length`, `xs:maxLength`, `xs:maxExclusive`, `xs:maxInclusive`, `xs:minExclusive`, `xs:minInclusive`, `xs:totalDigits` and `xs:fractionDigits` can be loosened or omitted.
- It is also possible to loosen restrictions by adding further `xs:pattern` facets, if there was already one `xs:pattern` which must remain unchanged.
- Fixed values may not be changed. Default values may be changed.
- Changes of the multiplicity are allowed if they loosen the schema. This means:.

  The `minOccurs` attribute for any `xs:element`, `xs:choice` or `xs:sequence` already described in the old schema may be decreased, whereas `maxOccurs` may be increased. Similarly, the `use` attribute of `xs:attribute` may be changed from "required" to "optional".

  If instances are to be stored that contain a node not yet described in the schema, the node can be added to the schema with `minOccurs`="0" (element) or with `use = "optional"` (attribute).

- Updating a schema can add, change or remove collations defined for elements or attributes.
- Doctypes can be added, but not removed. (See the section *Undefine from a Schema* below.)

Physical Schema

- The contents of the `tsd:index` element may be changed, except for objects mapped to a non-XML node.
- Attribute values may be changed, as long as this does not create conflicts (for example, a node's mapping type must not be changed).

  The `default` attributes of `xs:element` and `xs:attribute` elements may be changed.

Changing Element and Attribute Values in the Physical Schema

Generally, for all kinds of mapping, schema information may be changed, as long as this is compatible with existing objects.

For the sake of convenience, this section lists the attributes specifically related to the mapping possibilities for SQL, Adabas and Server Extensions.

SQLTable Mapping Information: the `tsd:subTreeSQL` element

The contents of the following child element of the `tsd:subTreeSQL` element may be changed:

```
tsd:accessPredicate
```

The values of the following attributes of the `tsd:subTreeSQL` element may be changed:

```
datasource
password
schema
table
userid
```

SQL Column Mapping Information: the `tsd:nodeSQL` element

The following attribute of the `tsd:nodeSQL` element may be changed:

```
column
```

Adabas File Mapping Information: the `subTreeAdabas` element

The following attributes of the `tsd:subTreeAdabas` element may be changed:

```
dbid
encoding
fnr
```

password

Adabas PE Mapping Information: the tsd:subTreeAdabasPE element

The following attribute of the tsd:subTreeAdabasPE element may be changed:

shortname

Adabas Field and Adabas MU Mapping Information: the tsd:nodeAdabasField element

The following attributes of the tsd:nodeAdabasField element can be changed:

encoding
format
length
shortname

Server Extension Function Mapping Information: the tsd:xTension element

All elements and attribute identifying the Server Extension function (tsd:xTension element) can be changed.

tsd:index Element

The contents of the tsd:index element may be changed. This means adding or removing support for standard or text indexing by adding or removing the tsd:standard or tsd:text element within the tsd:index element.

An exception is the indexing of non-XML documents: Currently, this cannot be switched on or off via update-define.

tsd:ignoreUpdate Element

The element tsd:ignoreUpdate may be added but not removed.

tsd:structureIndex Element

The node of the structure index for XML doctypes can be changed in an arbitrary fashion.

tsd:compress Element

The compression mode used for the document stored in a doctype may be changed via update-define. This does not change the compression of documents already stored in Tamino.

Schema Evolution for Open Content (Update Schema Processing)

Schema evolution (also called “update schema”) for open content differs from normal closed content. If you do not explicitly use `_mode=validate`, the following applies:

Unlike closed content, unknown elements with arbitrary content may exist. Therefore it is generally not possible to add elements to the schema for open content doctypes. On the other hand, removal of elements is possible without changing the integrity rule: all XML instances already stored in a doctype must remain valid with respect to the new schema.

The following rules apply for adding and removing elements from a content model:

1. Elements used in open content doctype: its CNS (child-element-name-set) cannot be extended.
2. Elements used in closed content doctype: optional child elements can be added
3. Elements used in both open and closed content doctypes: The conjunction of the two rules above holds: its CNS must be the same. This means only the multiplicity can be defined more loosely.
4. The doctype of elements used only in closed content is changed to open content: The disjunction of the two rules above results in the fact that the CNS can be reduced or increased.

Notes:

1. The open content *update schema* rules on the CNS are equivalent to the rules imposed for `xs:any` with `processContents="loose"` on global elements.
2. Analogous rules apply for attribute definitions.

Update Schema Checks for Imported Schemas

Prior to Tamino version 4.2, updating of schemas which had been imported using the `xs:import` element was not permitted.

Update Schema Checks for xsi:type

There are several scenarios in schema update where all documents containing `xsi:type` need to be revalidated. Revalidation can be requested via `_mode=validate`. Such cases occur for example:

- if global types have been added and there are wildcards with `processContents="lax"`;
- if global type definitions have been removed, causing references to them to become invalid.

Undefine

One or more doctypes can be removed from a TSD schema by sending the following command to Tamino:

```
_undefine = {doctype|schema|collectionname},...
```

The operand is a comma-separated list of `doctype`s, `schema`s and/or `collectionname`s, where:

- `doctype` is defined as *`collectionname`*/*`schemaname`*/ . . . /*`doctypename`*
- `schema` is defined as *`collectionname`*/*`schemaname`*

Undefining a doctype deletes it from Tamino, including all documents stored inside. The respective `tsd:doctype` element is removed from the schema document stored in Tamino. This may lead to the schema document not defining any doctype at all.

Deleting a schema deletes all doctypes that are defined in it.

There is a postcondition that dangling references to imported or included schemas must not exist after the undefine operation (referential integrity).

9 Tools for the Creation of Tamino Schemas

How you plan or define your Tamino schema depends very much on your starting position.

This chapter describes the options open to you under the following headings:

Defining a Schema from a DTD

You can start the process of defining a Tamino schema using an arbitrary external DTD.

Using the **Import DTD** function of the Tamino Schema Editor, you can read a DTD into the editor. You can then either use default mapping for the elements and attributes of the DTD, or you can specifically map each element and attribute that interests you.

A combination of both is also possible: you define default mapping and refine the mapping information for appropriate nodes, using the features of **XML Schema** that cannot be inferred from DTDs. Key **XML Schema** features not provided for by DTDs are:

- XML schemas are themselves XML documents and can therefore be parsed as such.
- Contents of elements and values of attributes can be specified as being of a specific datatype.
- The number of instances of element types within a parent element can be restricted.
- Text values in simple type elements or attributes can be restricted.
- The number of items in white-space separated lists that make up attribute values can be restricted.

Defining a Schema from a given XML Schema

You can start the process of defining a Tamino schema using an arbitrary external schema that conforms to the **XML Schema** standard.

You can read the XML schema into the Tamino Schema Editor and add Tamino-specific information to it.

Notes:

1. Tamino currently supports only a subset of **XML Schema** features. The schema editor supports full **XML Schema**, however.
2. For information about converting Tamino schemas from former versions, see the migration documentation of the current Tamino version and its predecessors.
3. At least the collection name and the schema name must be set, corresponding to the name attributes of `tsd:collection` and `tsd:schemaInfo`.

Defining a Schema from Scratch

You can define a Tamino schema from scratch, that is, you have neither an existing schema to migrate, nor an external DTD, nor an **XML Schema** as starting point for schema definition.

You can use the Tamino Schema Editor to build a Tamino schema. The advantage of using the Tamino Schema Editor over other schema editing tools is that the Tamino Schema Editor will not generate any syntax not supported by Tamino.

Of course, you can create a schema also using any arbitrary text editor. This, however, should be treated as error-prone.

Defining a Schema Using Third Party Products

You can also use a high-level tool from a third party vendor, for example XML Spy from Altova or <oXygen/> from SyncRO Soft Ltd., to create Tamino schemas.

A Appendices

The following table describes the combinations of datatypes and restricting facets that are allowed in TSD.

Simple datatype	length	minLength	maxLength	pattern	enumeration	whiteSpace	minExclusive	maxExclusive	minInclusi
anyURI	x	x	x	x	x	x			
base64Binary	x	x	x	x	x	x			
boolean				x		x			
byte				x	x	x	x	x	x
date				x	x	x	x	x	x
dateTime				x	x	x	x	x	x
decimal				x	x	x	x	x	x
double				x	x	x	x	x	x
duration				x	x	x	x	x	x
ENTITIES	x	x	x		x	x			
ENTITY	x	x	x	x	x	x			
float				x	x	x	x	x	x
gDay				x	x	x	x	x	x
gMonth				x	x	x	x	x	x
gMonthDay				x	x	x	x	x	x
gYear				x	x	x	x	x	x
gYearMonth				x	x	x	x	x	x
hexBinary	x	x	x	x	x	x			
ID	x	x	x	x	x	x			
IDREF	x	x	x	x	x	x			
IDREFS	x	x	x		x	x			
int				x	x	x	x	x	x

Simple datatype	length	minLength	maxLength	pattern	enumeration	whiteSpace	minExclusive	maxExclusive	minInclusive	maxIn
integer				x	x	x	x	x	x	x
language code (RFC 1766)	x	x	x	x	x	x				
long				x	x	x	x	x	x	x
Name	x	x	x	x	x	x				
NCName	x	x	x	x	x	x				
negativeInteger				x	x	x	x	x	x	x
NMTOKEN	x	x	x	x	x	x				
NMTOKENS	x	x	x		x	x				
nonNegativeInteger				x	x	x	x	x	x	x
nonPositiveInteger				x	x	x	x	x	x	x
normalizedString	x	x	x	x	x	x				
NOTATION	x	x	x	x	x	x				
positiveInteger				x	x	x	x	x	x	x
QName	x	x	x	x	x	x				
short				x	x	x	x	x	x	x
string	x	x	x	x	x	x				
time				x	x	x	x	x	x	x
token	x	x	x	x	x	x				
unsignedByte				x	x	x	x	x	x	x
unsignedInt				x	x	x	x	x	x	x
unsignedLong				x	x	x	x	x	x	x
unsignedShort				x	x	x	x	x	x	x

Combinations of Datatypes and Restricting Facets in TSD

B Appendices

The following language and country codes are available in TSD:

af	Afrikaans
af_ZA	Afrikaans (South Africa)
ar	Arabic
ar_AE	Arabic (United Arab Emirates)
ar_BH	Arabic (Bahrain)
ar_DZ	Arabic (Algeria)
ar_EG	Arabic (Egypt)
ar_IQ	Arabic (Iraq)
ar_JO	Arabic (Jordan)
ar_KW	Arabic (Kuwait)
ar_LB	Arabic (Lebanon)
ar_LY	Arabic (Libya)
ar_MA	Arabic (Morocco)
ar_OM	Arabic (Oman)
ar_QA	Arabic (Qatar)
ar_SA	Arabic (Saudi Arabia)
ar_SD	Arabic (Sudan)
ar_SY	Arabic (Syrian Arab Republic)
ar_TN	Arabic (Tunisia)
ar_YE	Arabic (Yemen)
be	Belorussian
be_BY	Belorussian (Belarus)
bg	Bulgarian

bg_BG	Bulgarian (Bulgaria)
ca	Catalan
ca_ES	Catalan (Spain)
ca_ES_EURO	Catalan (Spain, with Euro symbol)
cs	Czech
cs_CZ	Czech (Czech Republic)
da	Danish
da_DK	Danish (Denmark)
de	German
de__PHONEBOOK	German (Phonebook Sorting)
de_AT	German (Austria)
de_AT_EURO	German (Austria, with Euro symbol)
de_CH	German (Switzerland)
de_DE	German (Germany)
de_DE_EURO	German (Germany, with Euro symbol)
de_LU	German (Luxembourg)
de_LU_EURO	German (Luxembourg, with Euro symbol)
el	Greek
el_GR	Greek (Greece)
el_GR_EURO	Greek (Greece, with Euro symbol)
en	English
en_AS	English (American Samoa)
en_AU	English (Australia)
en_BE	English (Belgium)
en_BE_EURO	English (Belgium, with Euro symbol)
en_BW	English (Botswana)
en_CA	English (Canada)
en_GB	English (United Kingdom)
en_GB_EURO	English (United Kingdom, with Euro symbol)
en_GU	English (Guam)
en_HK	English (Hong Kong)
en_IE	English (Ireland)
en_IE_EURO	English (Ireland, with Euro symbol)
en_IN	English (India)
en_MH	English (Marshall Islands)
en_MP	English (Northern Mariana Islands)
en_NZ	English (New Zealand)

en_PH	English (Philippines)
en_SG	English (Singapore)
en_UM	English (United States Minor Outlying Islands)
en_US	English (United States)
en_US_POSIX	English (United States), POSIX-conforming
en_VI	English (Virgin Islands / U.S.)
en_ZA	English (South Africa)
en_ZW	English (Zimbabwe)
eo	Esperanto
es	Spanish
es__TRADITIONAL	Spanish (Spain)
es_AR	Spanish (Argentina)
es_BO	Spanish (Bolivia)
es_CL	Spanish (Chile)
es_CO	Spanish (Colombia)
es_CR	Spanish (Costa Rica)
es_DO	Spanish (Dominican Republic)
es_EC	Spanish (Ecuador)
es_ES	Spanish (Spain)
es_ES_EURO	Spanish (Spain, with Euro symbol)
es_GT	Spanish (Guatemala)
es_HN	Spanish (Honduras)
es_MX	Spanish (Mexico)
es_NI	Spanish (Nicaragua)
es_PA	Spanish (Panama)
es_PE	Spanish (Peru)
es_PR	Spanish (Puerto Rico)
es_PY	Spanish (Paraguay)
es_SV	Spanish (El Salvador)
es_US	Spanish (United States)
es_UY	Spanish (Uruguay)
es_VE	Spanish (Venezuela)
et	Estonian
et_EE	Estonian (Estonia)
eu	Basque
eu_ES	Basque (Spain)
eu_ES_EURO	Basque (Spain, with Euro symbol)

fa	Persian
fa_IR	Persian (Iran)
fi	Finnish
fi_FI	Finnish (Finland)
fi_FI_EURO	Finnish (Finland, with Euro symbol)
fo	Faroese
fo_FO	Faroese (Faroe Islands)
fr	French
fr_BE	French (Belgium)
fr_BE_EURO	French (Belgium, with Euro symbol)
fr_CA	French (Canada)
fr_CH	French (Switzerland)
fr_FR	French (France)
fr_FR_EURO	French (France, with Euro symbol)
fr_LU	French (Luxembourg)
fr_LU_EURO	French (Luxembourg, with Euro symbol)
ga	Irish (also called Gaelic)
ga_IE	Irish (Ireland)
ga_IE_EURO	Irish (Ireland, with Euro symbol)
gl	Galician
gl_ES	Galician (Spain)
gl_ES_EURO	Galician (Spain, with Euro symbol)
gv	
gv_GB	
he	Hebrew
he_IL	Hebrew (Israel)
hi	Hindi
hi__DIRECT	Hindi
hi_IN	Hindi (India)
hr	Croatian
hr_HR	Croatian (Croatia)
hu	Hungarian
hu_HU	Hungarian (Hungary)
id	Indonesian
id_ID	Icelandic (Indonesia)
is	Icelandic
is_IS	Icelandic (Iceland)

it	Italian
it_CH	Italian (Switzerland)
it_IT	Italian (Italy)
it_IT_EURO	Italian (Italy, with Euro symbol)
ja	Japanese
ja_JP	Japanese (Japan)
kl	Greenlandic
kl_GL	Greenlandic (Greenland)
ko	Korean
ko_KR	Korean (Republic Of Korea)
kok	
kok_IN	
kw	
kw_GB	
lt	Lithuanian
lt_LT	Lithuanian (Lithuania)
lv	Latvian
lv_LV	Latvian (Latvia)
mk	Macedonian
mk_MK	Macedonian (Macedonia)
mr	Marathi
mr_IN	Marathi (India)
mt	Maltese
mt_MT	Maltese (Malta)
nb	Norwegian / Bokmål
nb_NO	Norwegian / Bokmål (Norway)
nl	Dutch
nl_BE	Dutch (Belgium)
nl_BE_EURO	Dutch (Belgium, with Euro symbol)
nl_NL	Dutch (Netherlands)
nl_NL_EURO	Dutch (Netherlands, with Euro symbol)
nn	Norwegian / Nynorsk
nn_NO	Norwegian / Nynorsk (Norway)
pl	Polish
pl_PL	Polish (Poland)
pt	Portuguese
pt_BR	Portuguese (Brazil)

pt_PT	Portuguese (Portugal)
pt_PT_EURO	Portuguese (Portugal, with Euro symbol)
ro	Romanian
ro_RO	Romanian (Romania)
ru	Russian
ru_RU	Russian (Russian Federation)
ru_UA	Russian (Ukraine)
sh	Serbo-Croatian
sh_YU	Serbo-Croatian (Yugoslavia)
sk	Slovak
sk_SK	Slovak (Slovakia / Slovak Republic)
sl	Slovenian
sl_SI	Slovenian (Slovenia)
sq	Albanian
sq_AL	Albanian (Albania)
sr	Serbian
sr_YU	Serbian (Yugoslavia)
sv	Swedish
sv_FI	Swedish (Finland)
sv_FI_AL	Swedish (Åland Islands)
sv_SE	Swedish (Sweden)
sw	Swahili
sw_KE	Swahili (Kenya)
sw_TZ	Swahili (United Republic Of Tanzania)
ta	Tamil
ta_IN	Tamil (India)
te	Telugu
te_IN	Telugu (India)
th	Thai
th_TH	Thai (Thailand)
tr	Turkish
tr_TR	Turkish (Turkey)
uk	Ukrainian
uk_UA	Ukrainian (Ukraine)
vi	Vietnamese
vi_VN	Vietnamese (Vietnam)
zh	Chinese

zh__PINYIN	Chinese
zh_CN	Chinese (China)
zh_HK	Chinese (Hong Kong)
zh_SG	Chinese (Singapore)
zh_TW	Chinese (Taiwan)
zh_TW_STROKE	Chinese (Taiwan)

Available Language and Country Codes

C Appendices

This appendix contains the following examples:

Example 1: Storage of Whole Instance; Full Text Indexing for Selected Nodes

Specifying a `tsd:index` element with a `tsd:text` child element correctly stores the whole XML instance and performs full text indexing on the terminal nodes.

The definition of an index takes place in lines `0840-1130` for the elements “surname” and “first-name”, and in lines `1300-1600` for the elements `street` and `city`, respectively.

The *hospital* schema is as follows:

Note: Line numbers were added only to facilitate readability and reference.

```
0010 <?xml version = "1.0" encoding = "UTF-8"?>
0020 <xs:schema
 xmlns:xs = "http://www.w3.org/2001/XMLSchema"
 xmlns:tsd = "http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
0030   <xs:annotation>
0040     <xs:documentation>
0050     </xs:documentation>
0060     <xs:appinfo>
0070       <tsd:schemaInfo name = "patient"></tsd:schemaInfo>
0080     </xs:appinfo>
0090   </xs:annotation>
0100   <xs:element name = "patient">
0110     <xs:annotation>
0120       <xs:documentation> Elements </xs:documentation>
0130     </xs:annotation>
0140     <xs:complexType>
0150       <xs:sequence>
0160         <xs:element ref = "name" minOccurs="0"></xs:element>
0170         <xs:element ref = "sex"></xs:element>
0180         <xs:element ref = "born" minOccurs="0"></xs:element>
0190         <xs:element ref = "address" minOccurs="0"></xs:element>
0200         <xs:element ref = "occupation" minOccurs="0"></xs:element>
0210         <xs:element ref = "insurance" minOccurs="0"></xs:element>
0220         <xs:element ref = "nextofkin" minOccurs="0"></xs:element>
0230         <xs:element ref = "submitted"></xs:element>
0240         <xs:element ref = "examination" minOccurs="0" maxOccurs="unbounded"/>
0250         <xs:element ref = "therapy" minOccurs="0"></xs:element>
0260         <xs:element ref = "result" minOccurs="0"></xs:element>
0270         <xs:element ref = "remarks" minOccurs="0" maxOccurs="unbounded"/>
0280       </xs:sequence>
0290       <xs:attribute name = "id" type = "xs:ID">
0300         <xs:annotation>
0310           <xs:documentation> Attributes </xs:documentation>
```

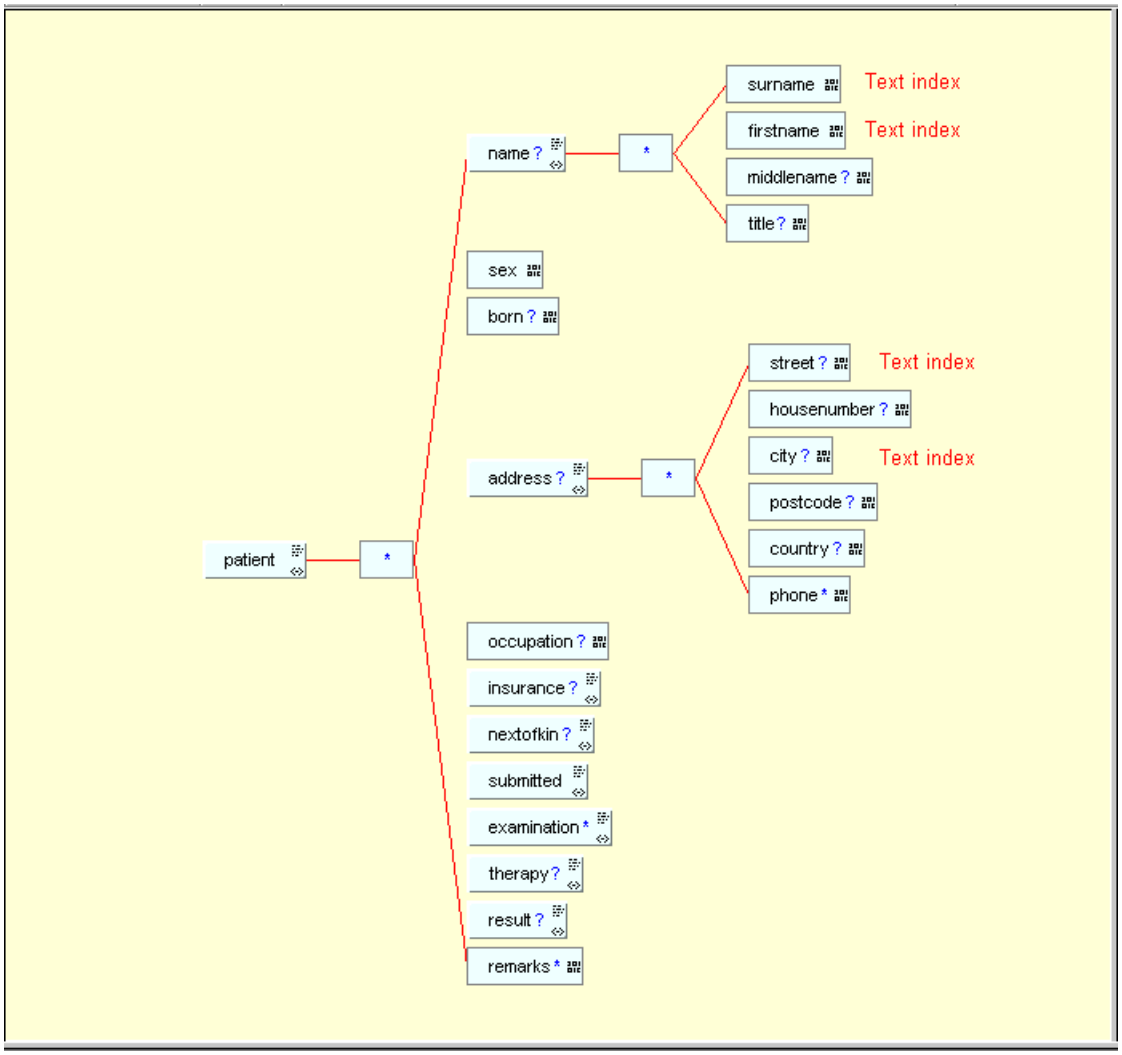
```
0320          </xs:annotation>
0330        </xs:attribute>
0340      </xs:complexType>
0350    </xs:element>
0360    <xs:element name = "submitted">
0370      <xs:complexType>
0380        <xs:sequence>
0390          <xs:element ref = "date"></xs:element>
0400          <xs:element ref = "symptoms"></xs:element>
0410          <xs:element ref = "diagnosis" maxOccurs="unbounded"/>
0420          <xs:element ref = "doctor" maxOccurs="unbounded"/>
0430        </xs:sequence>
0440      </xs:complexType>
0450    </xs:element>
0460    <xs:element name = "examination">
0470      <xs:complexType>
0480        <xs:sequence>
0490          <xs:element ref = "date"></xs:element>
0500          <xs:element ref = "report" minOccurs="0" maxOccurs="unbounded"/>
0510          <xs:element ref = "remarks" minOccurs="0" maxOccurs="unbounded"/>
0520        </xs:sequence>
0530      </xs:complexType>
0540    </xs:element>
0550    <xs:element name = "therapy">
0560      <xs:complexType>
0570        <xs:choice maxOccurs="unbounded">
0580          <xs:element ref = "medication"></xs:element>
0590          <xs:element ref = "physical"></xs:element>
0600          <xs:element ref = "other"></xs:element>
0610        </xs:choice>
0620      </xs:complexType>
0630    </xs:element>
0640    <xs:element name = "result">
0650      <xs:complexType>
0660        <xs:choice maxOccurs="unbounded">
0670          <xs:element ref = "dismissed"></xs:element>
0680          <xs:element ref = "transferred"></xs:element>
0690          <xs:element ref = "deceased"></xs:element>
0700        </xs:choice>
0710      </xs:complexType>
0720    </xs:element>
0730    <xs:element name = "remarks" type = "xs:string"></xs:element>
0740    <xs:element name = "name">
0750      <xs:complexType>
0760        <xs:sequence>
0770          <xs:element ref = "surname"></xs:element>
0780          <xs:element ref = "firstname"></xs:element>
0790          <xs:element ref = "middlename" minOccurs="0"></xs:element>
0800          <xs:element ref = "title" minOccurs="0"></xs:element>
0810        </xs:sequence>
0820      </xs:complexType>
0830    </xs:element>
```

```
0840   <xs:element name = "surname" type = "xs:string">
0850     <xs:annotation>
0860       <xs:appinfo>
0870         <tsd:elementInfo>
0880           <tsd:physical>
0890             <tsd:native>
0900               <tsd:index>
0910                 <tsd:text></tsd:text>
0920               </tsd:index>
0930             </tsd:native>
0940           </tsd:physical>
0950         </tsd:elementInfo>
0960       </xs:appinfo>
0970     </xs:annotation>
0980   </xs:element>
0990   <xs:element name = "firstname" type = "xs:string">
1000     <xs:annotation>
1010       <xs:appinfo>
1020         <tsd:elementInfo>
1030           <tsd:physical>
1040             <tsd:native>
1050               <tsd:index>
1060                 <tsd:text></tsd:text>
1070               </tsd:index>
1080             </tsd:native>
1090           </tsd:physical>
1100         </tsd:elementInfo>
1110       </xs:appinfo>
1120     </xs:annotation>
1130   </xs:element>
1140   <xs:element name = "middlename" type = "xs:string"></xs:element>
1150   <xs:element name = "title" type = "xs:string"></xs:element>
1160   <xs:element name = "sex" type = "xs:string"></xs:element>
1170   <xs:element name = "born" type = "xs:string"></xs:element>
1180   <xs:element name = "address">
1190     <xs:complexType>
1200       <xs:sequence>
1210         <xs:element ref = "street" minOccurs="0"></xs:element>
1220         <xs:element ref = "housenumber" minOccurs="0"></xs:element>
1230         <xs:element ref = "city" minOccurs="0"></xs:element>
1240         <xs:element ref = "postcode" minOccurs="0"></xs:element>
1250         <xs:element ref = "country" minOccurs="0"></xs:element>
1260         <xs:element ref = "phone" minOccurs="0" maxOccurs="unbounded"/>
1270       </xs:sequence>
1280     </xs:complexType>
1290   </xs:element>
1300   <xs:element name = "street" type = "xs:string">
1310     <xs:annotation>
1320       <xs:appinfo>
1330         <tsd:elementInfo>
1340           <tsd:physical>
1350             <tsd:native>
```

```
1360              <tsd:index>
1370                <tsd:text></tsd:text>
1380              </tsd:index>
1390            </tsd:native>
1400          </tsd:physical>
1410        </tsd:elementInfo>
1420      </xs:appinfo>
1430    </xs:annotation>
1440  </xs:element>
1450  <xs:element name = "housenumber" type = "xs:string"></xs:element>
1460  <xs:element name = "city" type = "xs:string">
1470    <xs:annotation>
1480      <xs:appinfo>
1490        <tsd:elementInfo>
1500          <tsd:physical>
1510            <tsd:native>
1520              <tsd:index>
1530                <tsd:text></tsd:text>
1540              </tsd:index>
1550            </tsd:native>
1560          </tsd:physical>
1570        </tsd:elementInfo>
1580      </xs:appinfo>
1590    </xs:annotation>
1600  </xs:element>
1610  <xs:element name = "postcode" type = "xs:string"></xs:element>
1620  <xs:element name = "country" type = "xs:string"></xs:element>
1630  <xs:element name = "occupation" type = "xs:string"></xs:element>
1640  <xs:element name = "insurance">
1650    <xs:complexType>
1660      <xs:sequence>
1670        <xs:element ref = "company"></xs:element>
1680        <xs:element ref = "policynumber" minOccurs="0"></xs:element>
1690      </xs:sequence>
1700    </xs:complexType>
1710  </xs:element>
1720  <xs:element name = "company" type = "xs:string"></xs:element>
1730  <xs:element name = "policynumber" type = "xs:string"></xs:element>
1740  <xs:element name = "nextofkin">
1750    <xs:complexType>
1760      <xs:sequence>
1770        <xs:element ref = "name"></xs:element>
1780        <xs:element ref = "address" minOccurs="0"></xs:element>
1790        <xs:element ref = "phone" minOccurs="0" maxOccurs="unbounded"/>
1800        <xs:element ref = "fax" minOccurs="0"></xs:element>
1810      </xs:sequence>
1820      <xs:attribute name = "grade" type = "xs:string" use = "required"/>
1830    </xs:complexType>
1840  </xs:element>
1850  <xs:element name = "phone" type = "xs:string"></xs:element>
1860  <xs:element name = "fax" type = "xs:string"></xs:element>
1870  <xs:element name = "date" type = "xs:string"></xs:element>
```

```
1880   <xs:element name = "report">
1890     <xs:complexType>
1900       <xs:attribute name = "files" type = "xs:ENTITIES" use = "required"/>
1910     </xs:complexType>
1920   </xs:element>
1930   <xs:element name = "symptoms" type = "xs:string"></xs:element>
1940   <xs:element name = "diagnosis" type = "xs:string"></xs:element>
1950   <xs:element name = "doctor">
1960     <xs:complexType>
1970       <xs:sequence>
1980         <xs:element ref = "name"></xs:element>
1990       </xs:sequence>
2000       <xs:attribute name = "pager" type = "xs:string"></xs:attribute>
2010     </xs:complexType>
2020   </xs:element>
2030   <xs:element name = "medication">
2040     <xs:complexType>
2050       <xs:sequence maxOccurs="unbounded">
2060         <xs:element ref = "type"></xs:element>
2070         <xs:element ref = "dosage"></xs:element>
2080       </xs:sequence>
2090     </xs:complexType>
2100   </xs:element>
2110   <xs:element name = "type">
2120     <xs:complexType>
2130       <xs:simpleContent>
2140         <xs:extension base = "xs:string">
2150           <xs:attribute name = "form" use = "required">
2160             <xs:simpleType>
2170               <xs:restriction base = "xs:NMTOKEN">
2180                 <xs:enumeration value = "tablet"></xs:enumeration>
2190                 <xs:enumeration value = "capsule"></xs:enumeration>
2200                 <xs:enumeration value = "drops"></xs:enumeration>
2210                 <xs:enumeration value = "intravenous"></xs:enumeration>
2220               </xs:restriction>
2230             </xs:simpleType>
2240           </xs:attribute>
2250           <xs:attribute name = "brand" type = "xs:string"></xs:attribute>
2260         </xs:extension>
2270       </xs:simpleContent>
2280     </xs:complexType>
2290   </xs:element>
2300   <xs:element name = "dosage" type = "xs:string"></xs:element>
2310   <xs:element name = "physical">
2320     <xs:complexType>
2330       <xs:sequence>
2340         <xs:element ref = "description"></xs:element>
2350         <xs:element ref = "frequency" minOccurs="0"></xs:element>
2360       </xs:sequence>
2370     </xs:complexType>
2380   </xs:element>
2390   <xs:element name = "description" type = "xs:string"></xs:element>
```

```
2400   <xs:element name = "frequency" type = "xs:string"></xs:element>
2410   <xs:element name = "other">
2420     <xs:complexType>
2430       <xs:sequence>
2440         <xs:element ref = "description"></xs:element>
2450         <xs:element ref = "amount" minOccurs="0"></xs:element>
2460       </xs:sequence>
2470     </xs:complexType>
2480   </xs:element>
2490   <xs:element name = "amount" type = "xs:string"></xs:element>
2500   <xs:element name = "dismissed">
2510     <xs:complexType>
2520       <xs:sequence>
2530         <xs:element ref = "date"></xs:element>
2540         <xs:element ref = "doctor" minOccurs="0"></xs:element>
2550       </xs:sequence>
2560     </xs:complexType>
2570   </xs:element>
2580   <xs:element name = "transferred">
2590     <xs:complexType>
2600       <xs:sequence>
2610         <xs:element ref = "date"></xs:element>
2620         <xs:element ref = "destination"></xs:element>
2630         <xs:element ref = "doctor" minOccurs="0"></xs:element>
2640       </xs:sequence>
2650     </xs:complexType>
2660   </xs:element>
2670   <xs:element name = "destination" type = "xs:string"></xs:element>
2680   <xs:element name = "deceased">
2690     <xs:complexType>
2700       <xs:sequence>
2710         <xs:element ref = "date"></xs:element>
2720         <xs:element ref = "doctor" minOccurs="0"></xs:element>
2730       </xs:sequence>
2740     </xs:complexType>
2750   </xs:element>
2760 </xs:schema>
```



Assuming the collection name is “hospital”, this makes specific queries based on the content of the terminal nodes most efficient, for example:

```
....../patient/name[surname~="A*"]
```

which returns all `name` elements that contain `surname` elements that start with “A”. Compare this expression with:

```
....../patient[name/surname~="A*"]
```

which returns all `patient` elements that contain a `name` element containing `surname` elements that start with "A".

Example 2: Hospital

This example comprises a whole XML schema that models patient data in a hospital application.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema"
           xmlns:tsd = ↩
"http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
  <xs:element name = "patient">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref = "name" minOccurs="0"></xs:element>
        <xs:element ref = "sex"></xs:element>
        <xs:element ref = "born" minOccurs="0"></xs:element>
        <xs:element ref = "address" minOccurs="0"></xs:element>
        <xs:element ref = "occupation" minOccurs="0"></xs:element>
        <xs:element ref = "insurance" minOccurs="0"></xs:element>
        <xs:element ref = "nextofkin" minOccurs="0"></xs:element>
        <xs:element ref = "submitted"></xs:element>
        <xs:element ref = "examination" minOccurs="0" ↩
maxOccurs="unbounded"></xs:element>
        <xs:element ref = "therapy" minOccurs="0"></xs:element>
        <xs:element ref = "result" minOccurs="0"></xs:element>
       <xs:element ref = "remarks" minOccurs="0" maxOccurs="unbounded"></xs:element>
      </xs:sequence>
      <xs:attribute name = "ID" type = "xs:string"></xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:element name = "submitted">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref = "date"></xs:element>
        <xs:element ref = "symptoms"></xs:element>
        <xs:element ref = "diagnosis" maxOccurs="unbounded"></xs:element>
        <xs:element ref = "doctor" maxOccurs="unbounded"></xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name = "examination">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref = "date"></xs:element>
       <xs:element ref = "report" minOccurs="0" maxOccurs="unbounded"></xs:element>
       <xs:element ref = "remarks" minOccurs="0" maxOccurs="unbounded"></xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name = "therapy">
```

```
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element ref = "medication"></xs:element>
        <xs:element ref = "physical"></xs:element>
        <xs:element ref = "other"></xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:element name = "result">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element ref = "discharged"></xs:element>
        <xs:element ref = "transferred"></xs:element>
        <xs:element ref = "deceased"></xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:element name = "remarks" type = "xs:string"></xs:element>
  <xs:element name = "name">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref = "surname"></xs:element>
        <xs:element ref = "firstname"></xs:element>
        <xs:element ref = "middlename" minOccurs="0"></xs:element>
        <xs:element ref = "title" minOccurs="0"></xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name = "surname" type = "xs:string"></xs:element>
  <xs:element name = "firstname" type = "xs:string"></xs:element>
  <xs:element name = "middlename" type = "xs:string"></xs:element>
  <xs:element name = "title" type = "xs:string"></xs:element>
  <xs:element name = "sex" type = "xs:string"></xs:element>
  <xs:element name = "born" type = "xs:string"></xs:element>
  <xs:element name = "address">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref = "street" minOccurs="0"></xs:element>
        <xs:element ref = "housenumber" minOccurs="0"></xs:element>
        <xs:element ref = "city" minOccurs="0"></xs:element>
        <xs:element ref = "postcode" minOccurs="0"></xs:element>
        <xs:element ref = "country" minOccurs="0"></xs:element>
       <xs:element ref = "phone" minOccurs="0" maxOccurs="unbounded"></xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name = "street" type = "xs:string"></xs:element>
  <xs:element name = "housenumber" type = "xs:string"></xs:element>
  <xs:element name = "city" type = "xs:string"></xs:element>
  <xs:element name = "postcode" type = "xs:string"></xs:element>
  <xs:element name = "country" type = "xs:string"></xs:element>
  <xs:element name = "occupation" type = "xs:string"></xs:element>
```

```
<xs:element name = "insurance">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref = "company"></xs:element>
      <xs:element ref = "policynumber" minOccurs="0"></xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name = "company" type = "xs:string"></xs:element>
<xs:element name = "policynumber" type = "xs:string"></xs:element>
<xs:element name = "nextofkin">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref = "name"></xs:element>
      <xs:element ref = "address" minOccurs="0"></xs:element>
     <xs:element ref = "phone" minOccurs="0" maxOccurs="unbounded"></xs:element>
      <xs:element ref = "fax" minOccurs="0"></xs:element>
    </xs:sequence>
    <xs:attribute name = "grade" type = "xs:string"></xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name = "phone" type = "xs:string"></xs:element>
<xs:element name = "fax" type = "xs:string"></xs:element>
<xs:element name = "date" type = "xs:string"></xs:element>
<xs:element name = "report">
  <xs:complexType>
   <xs:attribute name = "files" type = "xs:string" use = "required"></xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name = "symptoms" type = "xs:string"></xs:element>
<xs:element name = "diagnosis" type = "xs:string"></xs:element>
<xs:element name = "doctor">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref = "name"></xs:element>
    </xs:sequence>
    <xs:attribute name = "doctor" type = "xs:string"></xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name = "medication">
  <xs:complexType>
    <xs:sequence maxOccurs="unbounded">
      <xs:element ref = "type"></xs:element>
      <xs:element ref = "dosage"></xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name = "type">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base = "xs:string">
        <xs:attribute name = "form" type = "xs:string" use = ↩
```

```
"required"></xs:attribute>
          <xs:attribute name = "brand" type = "xs:string"></xs:attribute>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name = "dosage" type = "xs:string"></xs:element>
  <xs:element name = "physical">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref = "description"></xs:element>
        <xs:element ref = "frequency" minOccurs="0"></xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name = "description" type = "xs:string"></xs:element>
  <xs:element name = "frequency" type = "xs:string"></xs:element>
  <xs:element name = "other">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref = "description"></xs:element>
        <xs:element ref = "amount" minOccurs="0"></xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name = "amount" type = "xs:string"></xs:element>
  <xs:element name = "discharged">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref = "date"></xs:element>
        <xs:element ref = "doctor" minOccurs="0"></xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name = "transferred">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref = "date"></xs:element>
        <xs:element ref = "destination"></xs:element>
        <xs:element ref = "doctor" minOccurs="0"></xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name = "destination" type = "xs:string"></xs:element>
  <xs:element name = "deceased">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref = "date"></xs:element>
        <xs:element ref = "doctor" minOccurs="0"></xs:element>
      </xs:sequence>
    </xs:complexType>
```

```
  </xs:element>
</xs:schema>
```

Example 3: Telephone

This is a complete XML schema according to the telephone example used at some places elsewhere in this documentation:

```
<?xml version = "1.0" encoding = "euc-kr"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema"
           xmlns:tsd = ↩
"http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
  <xs:annotation>
    <xs:appinfo>
      <tsd:schemaInfo name = "Telephone">
        <tsd:collection name = "Sample"></tsd:collection>
        <tsd:doctype name = "Telephone">
          <tsd:logical>
            <tsd:content>closed</tsd:content>
          </tsd:logical>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name = "Telephone">
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "LoginName" type = "xs:string">
          <xs:annotation>
            <xs:appinfo>
              <tsd:elementInfo>
                <tsd:physical>
                  <tsd:native>
                    <tsd:index>
                      <tsd:standard></tsd:standard>
                    </tsd:index>
                  </tsd:native>
                </tsd:physical>
              </tsd:elementInfo>
            </xs:appinfo>
          </xs:annotation>
        </xs:element>
        <xs:element name = "PassWord" type = "xs:string">
          <xs:annotation>
            <xs:appinfo>
              <tsd:elementInfo>
                <tsd:physical>
                  <tsd:native>
```

```
                <tsd:index>
                  <tsd:standard></tsd:standard>
                </tsd:index>
              </tsd:native>
            </tsd:physical>
          </tsd:elementInfo>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
    <xs:element name = "Lastname" type = "xs:string">
      <xs:annotation>
        <xs:appinfo>
          <tsd:elementInfo>
            <tsd:physical>
              <tsd:native>
                <tsd:index>
                  <tsd:text></tsd:text>
                </tsd:index>
              </tsd:native>
            </tsd:physical>
          </tsd:elementInfo>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
    <xs:element name = "Firstname" type = "xs:string">
      <xs:annotation>
        <xs:appinfo>
          <tsd:elementInfo>

            <tsd:physical>
              <tsd:native>
                <tsd:index>
                  <tsd:standard></tsd:standard>
                </tsd:index>
              </tsd:native>
            </tsd:physical>
          </tsd:elementInfo>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
    <xs:element name = "Date_of_Birth" type = "xs:string">
      <xs:annotation>
        <xs:appinfo>
          <tsd:elementInfo>
            <tsd:physical>
              <tsd:native>
                <tsd:index>
                  <tsd:standard></tsd:standard>
                </tsd:index>
              </tsd:native>
            </tsd:physical>
          </tsd:elementInfo>
```

```
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
    <xs:element name = "Company_Name" type = "xs:string">
      <xs:annotation>
        <xs:appinfo>
          <tsd:elementInfo>
            <tsd:physical>
              <tsd:native>
                <tsd:index>
                  <tsd:standard></tsd:standard>
                </tsd:index>
              </tsd:native>
            </tsd:physical>
          </tsd:elementInfo>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
    <xs:element name = "Salutation" type = "xs:string">
      <xs:annotation>
        <xs:appinfo>
          <tsd:elementInfo>
            <tsd:physical>
              <tsd:native>
                <tsd:index>
                  <tsd:standard></tsd:standard>
                </tsd:index>
              </tsd:native>
            </tsd:physical>
          </tsd:elementInfo>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
    <xs:element name = "Email" type = "xs:string">
      <xs:annotation>
        <xs:appinfo>
          <tsd:elementInfo>
            <tsd:physical>
              <tsd:native>
                <tsd:index>
                  <tsd:standard></tsd:standard>
                </tsd:index>
              </tsd:native>
            </tsd:physical>
          </tsd:elementInfo>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
    <xs:element name = "Address">
      <xs:complexType>
        <xs:sequence>
          <xs:element name = "Street" type = "xs:string">
```

```
        <xs:annotation>
          <xs:appinfo>
            <tsd:elementInfo>
              <tsd:physical>
                <tsd:native>
                  <tsd:index>
                    <tsd:standard></tsd:standard>
                  </tsd:index>
                </tsd:native>
              </tsd:physical>
            </tsd:elementInfo>
          </xs:appinfo>
        </xs:annotation>
      </xs:element>
      <xs:element name = "City" type = "xs:string">
        <xs:annotation>
          <xs:appinfo>
            <tsd:elementInfo>
              <tsd:physical>
                <tsd:native>
                  <tsd:index>
                    <tsd:standard></tsd:standard>
                  </tsd:index>
                </tsd:native>
              </tsd:physical>
            </tsd:elementInfo>
          </xs:appinfo>
        </xs:annotation>
      </xs:element>
      <xs:element name = "ZIP" type = "xs:integer">
        <xs:annotation>
          <xs:appinfo>
            <tsd:elementInfo>
              <tsd:physical>
                <tsd:native>
                  <tsd:index>
                    <tsd:standard></tsd:standard>
                  </tsd:index>
                </tsd:native>
              </tsd:physical>
            </tsd:elementInfo>
          </xs:appinfo>
        </xs:annotation>
      </xs:element>
      <xs:element name = "Country" type = "xs:string">
        <xs:annotation>
          <xs:appinfo>
            <tsd:elementInfo>
              <tsd:physical>
                <tsd:native>
                  <tsd:index>
                    <tsd:standard></tsd:standard>
```

```
                    </tsd:index>
                  </tsd:native>
                </tsd:physical>
              </tsd:elementInfo>
            </xs:appinfo>
          </xs:annotation>
        </xs:element>
        <xs:element name = "Phone" type = "xs:string">
          <xs:annotation>
            <xs:appinfo>
              <tsd:elementInfo>
                <tsd:physical>
                  <tsd:native>
                    <tsd:index>
                      <tsd:standard></tsd:standard>
                    </tsd:index>
                  </tsd:native>
                </tsd:physical>
              </tsd:elementInfo>
            </xs:appinfo>
          </xs:annotation>
        </xs:element>
        <xs:element name = "Fax" type = "xs:string">
          <xs:annotation>
            <xs:appinfo>
              <tsd:elementInfo>
                <tsd:physical>
                  <tsd:native>
                    <tsd:index>
                      <tsd:standard></tsd:standard>
                    </tsd:index>
                  </tsd:native>
                </tsd:physical>
              </tsd:elementInfo>
            </xs:appinfo>
          </xs:annotation>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:sequence>
<xs:attribute name = "EntryID" type = "xs:integer">
  <xs:annotation>
    <xs:appinfo>
      <tsd:attributeInfo>
        <tsd:physical>
          <tsd:native>
            <tsd:index>
              <tsd:standard></tsd:standard>
            </tsd:index>
          </tsd:native>
        </tsd:physical>
```

```
            </tsd:attributeInfo>
          </xs:appinfo>
        </xs:annotation>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

D Appendix

This section contains some examples of schemas that define various different kinds of indices. The following cases are shown here:

Example 1: Simple Text Indexing

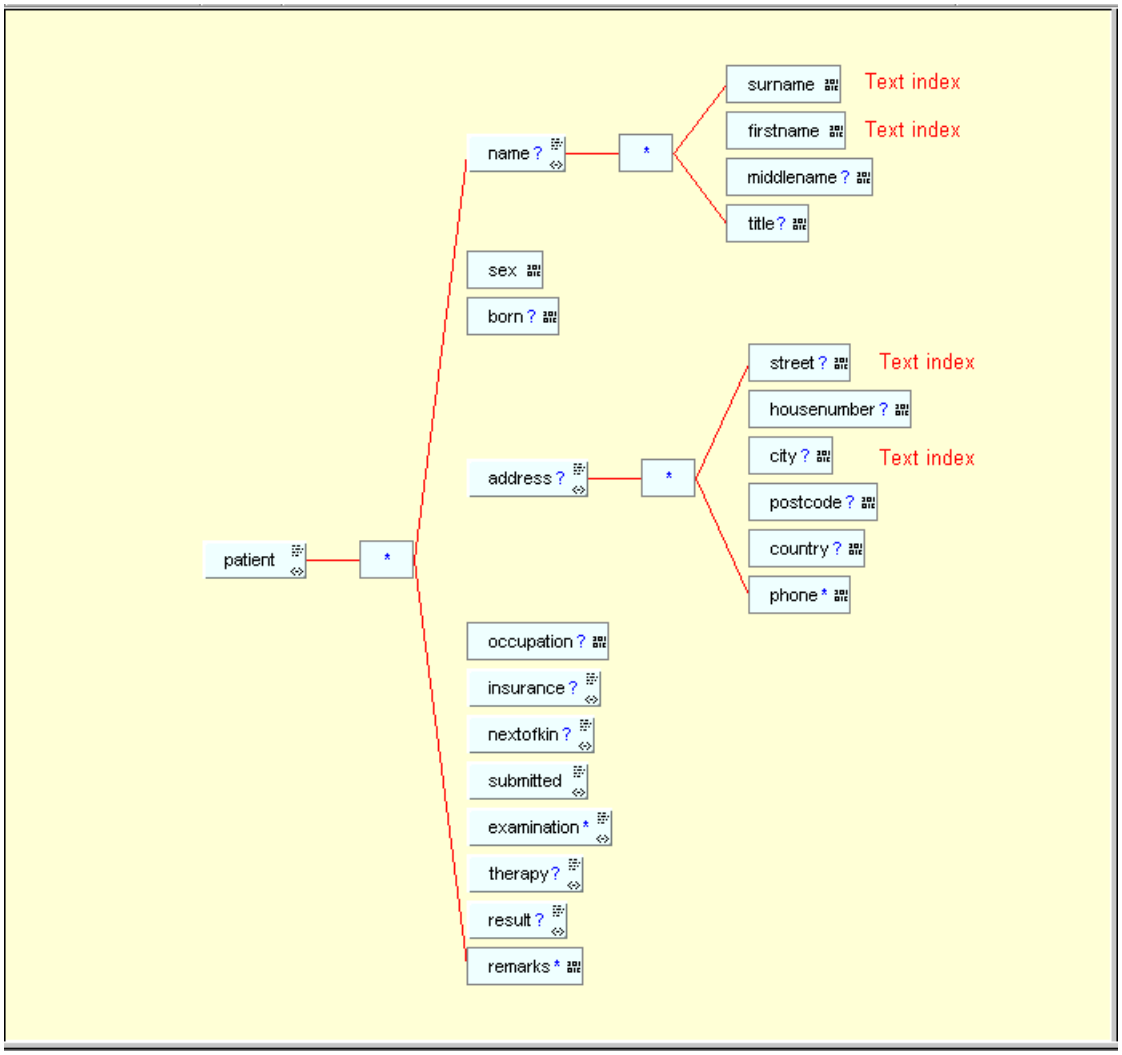
The following example illustrates how a simple Tamino query can be optimized by defining an appropriate index. For a full description of the query language, see the X-Query User Guide.

The example illustrate how the efficiency of retrieval operations can be influenced by intelligent indexing.

Specifying a `tsd:index` element with a `tsd:text` child element correctly stores the whole XML instance and performs full text indexing on the terminal nodes.

The *hospital* example is structured as shown in the picture:

Note: The source of the *hospital* example is to be found in the section ***Examples*** of the appendix. Line numbers were added only for improving readability and easy reference.



This picture shows the definition of the `patient` element within the *hospital* / `patient` schema example.

The corresponding schema looks like this:

Note: The definition of an index takes place in the lines of code from `0870` to `1160` and also the lines `1340` to `1640`:

```
0010 <?xml version="1.0" encoding="utf-8"?>
0020 <xs:schema>
0030   <xs:annotation>
0040     <xs:documentation></xs:documentation>
0050     <xs:appinfo>
0060       <tsd:schemaInfo name="patient"></tsd:schemaInfo>
0070     </xs:appinfo>
0080   </xs:annotation>
0090   <xs:element name="patient">
```

```
0100      <xs:annotation>
0110        <xs:documentation>Elements</xs:documentation>
0120      </xs:annotation>
0130      <xs:complexType>
0140        <xs:sequence>
0150          <xs:element ref="name" minOccurs="0"></xs:element>
0160          <xs:element ref="sex"></xs:element>
0170          <xs:element ref="born" minOccurs="0"></xs:element>
0180          <xs:element ref="address" minOccurs="0"></xs:element>
0190          <xs:element ref="occupation" minOccurs="0"></xs:element>
0200          <xs:element ref="insurance" minOccurs="0"></xs:element>
0210          <xs:element ref="nextofkin" minOccurs="0"></xs:element>
0220          <xs:element ref="submitted"></xs:element>
0230          <xs:element ref="examination" minOccurs="0" maxOccurs="unbounded">
0240          </xs:element>
0250          <xs:element ref="therapy" minOccurs="0"></xs:element>
0260          <xs:element ref="result" minOccurs="0"></xs:element>
0270          <xs:element ref="remarks" minOccurs="0" maxOccurs="unbounded">
0280          </xs:element>
0290        </xs:sequence>
0300        <xs:attribute name="id" type="xs:ID">
0310          <xs:annotation>
0320            <xs:documentation>Attributes</xs:documentation>
0330          </xs:annotation>
0340        </xs:attribute>
0350      </xs:complexType>
0360    </xs:element>
0370    <xs:element name="submitted">
0380      <xs:complexType>
0390        <xs:sequence>
0400          <xs:element ref="date"></xs:element>
0410          <xs:element ref="symptoms"></xs:element>
0420          <xs:element ref="diagnosis" maxOccurs="unbounded"></xs:element>
0430          <xs:element ref="doctor" maxOccurs="unbounded"></xs:element>
0440        </xs:sequence>
0450      </xs:complexType>
0460    </xs:element>
0470    <xs:element name="examination">
0480      <xs:complexType>
0490        <xs:sequence>
0500          <xs:element ref="date"></xs:element>
0510          <xs:element ref="report" minOccurs="0" maxOccurs="unbounded">
0520          </xs:element>
0530          <xs:element ref="remarks" minOccurs="0" maxOccurs="unbounded">
0540          </xs:element>
0550        </xs:sequence>
0560      </xs:complexType>
0570    </xs:element>
0580    <xs:element name="therapy">
0590      <xs:complexType>
0600        <xs:choice maxOccurs="unbounded">
0610          <xs:element ref="medication"></xs:element>
```

```
0620          <xs:element ref="physical"></xs:element>
0630          <xs:element ref="other"></xs:element>
0640        </xs:choice>
0650      </xs:complexType>
0660    </xs:element>
0670    <xs:element name="result">
0680      <xs:complexType>
0690        <xs:choice maxOccurs="unbounded">
0700          <xs:element ref="dismissed"></xs:element>
0710          <xs:element ref="transferred"></xs:element>
0720          <xs:element ref="deceased"></xs:element>
0730        </xs:choice>
0740      </xs:complexType>
0750    </xs:element>
0760    <xs:element name="remarks" type="xs:string"></xs:element>
0770    <xs:element name="name">
0780      <xs:complexType>
0790        <xs:sequence>
0800          <xs:element ref="surname"></xs:element>
0810          <xs:element ref="firstname"></xs:element>
0820          <xs:element ref="middlename" minOccurs="0"></xs:element>
0830          <xs:element ref="title" minOccurs="0"></xs:element>
0840        </xs:sequence>
0850      </xs:complexType>
0860    </xs:element>
0870    <xs:element name="surname" type="xs:string">
0880      <xs:annotation>
0890        <xs:appinfo>
0900          <tsd:elementInfo>
0910            <tsd:physical>
0920              <tsd:native>
0930                <tsd:index>
0940                  <tsd:text></tsd:text>
0950                </tsd:index>
0960              </tsd:native>
0970            </tsd:physical>
0980          </tsd:elementInfo>
0990        </xs:appinfo>
1000      </xs:annotation>
1010    </xs:element>
1020    <xs:element name="firstname" type="xs:string">
1030      <xs:annotation>
1040        <xs:appinfo>
1050          <tsd:elementInfo>
1060            <tsd:physical>
1070              <tsd:native>
1080                <tsd:index>
1090                  <tsd:text></tsd:text>
1100                </tsd:index>
1110              </tsd:native>
1120            </tsd:physical>
1130          </tsd:elementInfo>
```

```
1140        </xs:appinfo>
1150      </xs:annotation>
1160    </xs:element>
1170    <xs:element name="middlename" type="xs:string"></xs:element>
1180    <xs:element name="title" type="xs:string"></xs:element>
1190    <xs:element name="sex" type="xs:string"></xs:element>
1200    <xs:element name="born" type="xs:string"></xs:element>
1210    <xs:element name="address">
1220      <xs:complexType>
1230        <xs:sequence>
1240          <xs:element ref="street" minOccurs="0"></xs:element>
1250          <xs:element ref="housenumber" minOccurs="0"></xs:element>
1260          <xs:element ref="city" minOccurs="0"></xs:element>
1270          <xs:element ref="postcode" minOccurs="0"></xs:element>
1280          <xs:element ref="country" minOccurs="0"></xs:element>
1290          <xs:element ref="phone" minOccurs="0" maxOccurs="unbounded">
1300          </xs:element>
1310        </xs:sequence>
1320      </xs:complexType>
1330    </xs:element>
1340    <xs:element name="street" type="xs:string">
1350      <xs:annotation>
1360        <xs:appinfo>
1370          <tsd:elementInfo>
1380            <tsd:physical>
1390              <tsd:native>
1400                <tsd:index>
1410                  <tsd:text></tsd:text>
1420                </tsd:index>
1430              </tsd:native>
1440            </tsd:physical>
1450          </tsd:elementInfo>
1460        </xs:appinfo>
1470      </xs:annotation>
1480    </xs:element>
1490    <xs:element name="housenumber" type="xs:string"></xs:element>
1500    <xs:element name="city" type="xs:string">
1510      <xs:annotation>
1520        <xs:appinfo>
1530          <tsd:elementInfo>
1540            <tsd:physical>
1550              <tsd:native>
1560                <tsd:index>
1570                  <tsd:text></tsd:text>
1580                </tsd:index>
1590              </tsd:native>
1600            </tsd:physical>
1610          </tsd:elementInfo>
1620        </xs:appinfo>
1630      </xs:annotation>
1640    </xs:element>
1650    <xs:element name="postcode" type="xs:string"></xs:element>
```

```
1660   <xs:element name="country" type="xs:string"></xs:element>
1670   <xs:element name="occupation" type="xs:string"></xs:element>
1680   <xs:element name="insurance">
1690     <xs:complexType>
1700       <xs:sequence>
1710         <xs:element ref="company"></xs:element>
1720         <xs:element ref="policynumber" minOccurs="0"></xs:element>
1730       </xs:sequence>
1740     </xs:complexType>
1750   </xs:element>
1760   <xs:element name="company" type="xs:string"></xs:element>
1770   <xs:element name="policynumber" type="xs:string"></xs:element>
1780   <xs:element name="nextofkin">
1790     <xs:complexType>
1800       <xs:sequence>
1810         <xs:element ref="name"></xs:element>
1820         <xs:element ref="address" minOccurs="0"></xs:element>
1830         <xs:element ref="phone" minOccurs="0" maxOccurs="unbounded">
1840         </xs:element>
1850         <xs:element ref="fax" minOccurs="0"></xs:element>
1860       </xs:sequence>
1870       <xs:attribute name="grade" type="xs:string" use="required">
1880       </xs:attribute>
1890     </xs:complexType>
1900   </xs:element>
1910   <xs:element name="phone" type="xs:string"></xs:element>
1920   <xs:element name="fax" type="xs:string"></xs:element>
1930   <xs:element name="date" type="xs:string"></xs:element>
1940   <xs:element name="report">
1950     <xs:complexType>
1960       <xs:attribute name="files" type="xs:ENTITIES" use="required">
1970       </xs:attribute>
1980     </xs:complexType>
1990   </xs:element>
2000   <xs:element name="symptoms" type="xs:string"></xs:element>
2010   <xs:element name="diagnosis" type="xs:string"></xs:element>
2020   <xs:element name="doctor">
2030     <xs:complexType>
2040       <xs:sequence>
2050         <xs:element ref="name"></xs:element>
2060       </xs:sequence>
2070       <xs:attribute name="pager" type="xs:string"></xs:attribute>
2080     </xs:complexType>
2090   </xs:element>
2100   <xs:element name="medication">
2110     <xs:complexType>
2120       <xs:sequence maxOccurs="unbounded">
2130         <xs:element ref="type"></xs:element>
2140         <xs:element ref="dosage"></xs:element>
2150       </xs:sequence>
2160     </xs:complexType>
2170   </xs:element>
```

```
2180   <xs:element name="type">
2190     <xs:complexType>
2200       <xs:simpleContent>
2210         <xs:extension base="xs:string">
2220           <xs:attribute name="form" use="required">
2230             <xs:simpleType>
2240               <xs:restriction base="xs:NMTOKEN">
2250                 <xs:enumeration value="tablet"></xs:enumeration>
2260                 <xs:enumeration value="capsule"></xs:enumeration>
2270                 <xs:enumeration value="drops"></xs:enumeration>
2280                 <xs:enumeration value="intravenous"></xs:enumeration>
2290               </xs:restriction>
2300             </xs:simpleType>
2310           </xs:attribute>
2320           <xs:attribute name="brand" type="xs:string"></xs:attribute>
2330         </xs:extension>
2340       </xs:simpleContent>
2350     </xs:complexType>
2360   </xs:element>
2370   <xs:element name="dosage" type="xs:string"></xs:element>
2380   <xs:element name="physical">
2390     <xs:complexType>
2400       <xs:sequence>
2410         <xs:element ref="description"></xs:element>
2420         <xs:element ref="frequency" minOccurs="0"></xs:element>
2430       </xs:sequence>
2440     </xs:complexType>
2450   </xs:element>
2460   <xs:element name="description" type="xs:string"></xs:element>
2470   <xs:element name="frequency" type="xs:string"></xs:element>
2480   <xs:element name="other">
2490     <xs:complexType>
2500       <xs:sequence>
2510         <xs:element ref="description"></xs:element>
2520         <xs:element ref="amount" minOccurs="0"></xs:element>
2530       </xs:sequence>
2540     </xs:complexType>
2550   </xs:element>
2560   <xs:element name="amount" type="xs:string"></xs:element>
2570   <xs:element name="dismissed">
2580     <xs:complexType>
2590       <xs:sequence>
2600         <xs:element ref="date"></xs:element>
2610         <xs:element ref="doctor" minOccurs="0"></xs:element>
2620       </xs:sequence>
2630     </xs:complexType>
2640   </xs:element>
2650   <xs:element name="transferred">
2660     <xs:complexType>
2670       <xs:sequence>
2680         <xs:element ref="date"></xs:element>
2690         <xs:element ref="destination"></xs:element>
```

```
2700         <xs:element ref="doctor" minOccurs="0"></xs:element>
2710       </xs:sequence>
2720     </xs:complexType>
2730   </xs:element>
2740   <xs:element name="destination" type="xs:string"></xs:element>
2750   <xs:element name="deceased">
2760     <xs:complexType>
2770       <xs:sequence>
2780         <xs:element ref="date"></xs:element>
2790         <xs:element ref="doctor" minOccurs="0"></xs:element>
2800       </xs:sequence>
2810     </xs:complexType>
2820   </xs:element>
2830 </xs:schema>
```

Example 2: Simple Standard Indexing

Imagine the schema of the example above would have been changed by the following schema fragment concerning the definition of a standard index for the element called `surname` of type `xs:string`:

```
      .
      .
      .
      <xs:element name = "surname" type = "xs:string">
        <xs:annotation>
          <xs:appinfo>
            <tsd:elementInfo>
              <tsd:physical>
                <tsd:native>
                  <tsd:index>
                    <tsd:standard/>
                  </tsd:index>
                </tsd:native>
              </tsd:physical>
            </tsd:elementInfo>
          </xs:appinfo>
        </xs:annotation>
      </xs:element>
      .
      .
      .
```

This would lead to the creation of a standard index instead of a text index for the element `surname`.

See also the description of `tsd:index`.

Example 3: Defining a Compound Index

This example schema provides four elements located on 3 different levels, whereby A and B are of multiplicity greater than 1:

1. An element A one stage below root level
2. A child element of A named B
3. Two child elements of B named C and D.

On B, a compound index is declared for the children C and D which are of datatype “string” and of datatype “integer”, respectively.

Note: The definition of a compound index takes place in the lines of code from 250 to 300.

```
010 <?xml version="1.0" encoding="utf-8"?>
020 <xs:schema ↩
xmlns:tsd="http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition"
030           xmlns:xs="http://www.w3.org/2001/XMLSchema">
040   <xs:annotation>
050     <xs:appinfo>
060       <tsd:schemaInfo name="DocTreeIndex">
070         <tsd:collection name="DocTreeIndex"></tsd:collection>
080         <tsd:doctype name="Doc" />
090         <tsd:doctype name="Doc1" />
100       </tsd:schemaInfo>
110     </xs:appinfo>
120   </xs:annotation>
130   <xs:element name="Doc">
140     <xs:complexType>
150       <xs:sequence>
160         <xs:element name="A" maxOccurs="unbounded">
170           <xs:complexType>
180             <xs:sequence>
190               <xs:element name="B" maxOccurs="unbounded">
200                 <xs:annotation>
210                   <xs:appinfo>
220                     <tsd:elementInfo>
230                       <tsd:physical>
240                         <tsd:native>
250                           <tsd:index>
260                             <tsd:standard>
270                               <tsd:field xpath="C" />
280                               <tsd:field xpath="D" />
290                             </tsd:standard>
300                           </tsd:index>
310                         </tsd:native>
```

```
320                       </tsd:physical>
330                     </tsd:elementInfo>
340                   </xs:appinfo>
350                 </xs:annotation>
360                 <xs:complexType>
370                   <xs:sequence>
380                     <xs:element name="C" type="xs:string"></xs:element>
390                     <xs:element name="D" type="xs:integer"></xs:element>
400                   </xs:sequence>
410                 </xs:complexType>
420               </xs:element>
430             </xs:sequence>
440           </xs:complexType>
450         </xs:element>
460         <xs:element name="refDoc1" type="xs:string"></xs:element>
470       </xs:sequence>
480     </xs:complexType>
490   </xs:element>
500   <xs:element name="Doc1">
510     <xs:complexType>
520       <xs:sequence>
530         <xs:element name="A1" type="xs:string">
540           <xs:annotation>
550             <xs:appinfo>
560               <tsd:elementInfo>
570                 <tsd:physical>
580                   <tsd:native>
590                     <tsd:index>
600                       <tsd:standard></tsd:standard>
610                     </tsd:index>
620                   </tsd:native>
630                 </tsd:physical>
640               </tsd:elementInfo>
650             </xs:appinfo>
660           </xs:annotation>
670         </xs:element>
680         <xs:element name="F" type="xs:string"></xs:element>
690       </xs:sequence>
700     </xs:complexType>
710   </xs:element>
720 </xs:schema>
```

Example 4: Defining a Multipath Index

The schema must provide the path `/Doc/A/E` to the node `F`. The node `E` has a recursive structure `E->E`. `F` has a multiple path index of type standard with datatype `xs:string`. Below the node `A` as well the sub-tree `B` with `C` and `D` are defined. `A` and `B` are without multiplicity greater than 1.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:tsd="http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:annotation>
    <xs:appinfo>
      <tsd:schemaInfo name="DocRecursive">
        <tsd:collection name="DocRecursive"></tsd:collection>
        <tsd:doctype name="Doc">
          <tsd:logical>
            <tsd:content>closed</tsd:content>
          </tsd:logical>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name="Doc">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="A">
          <xs:complexType>
            <xs:sequence>
              <xs:element ref="E"></xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="B">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="C" type="xs:string">
                <xs:annotation>
                  <xs:appinfo>
                    <tsd:elementInfo>
                      <tsd:physical>
                        <tsd:native>
                          <tsd:index>
                            <tsd:standard></tsd:standard>
                          </tsd:index>
                        </tsd:native>
                      </tsd:physical>
                    </tsd:elementInfo>
                  </xs:appinfo>
                </xs:annotation>
```

```
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="E">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="E" minOccurs="0"></xs:element>
    <xs:element name="F" type="xs:string">
     <xs:annotation>
      <xs:appinfo>
       <tsd:elementInfo>
        <tsd:physical>
         <tsd:native>
          <tsd:index>
           <tsd:standard>
            <tsd:multiPath />
           </tsd:standard>
          </tsd:index>
         </tsd:native>
        </tsd:physical>
       </tsd:elementInfo>
      </xs:appinfo>
     </xs:annotation>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Example 5: Combining Various Indexing Techniques at One Single Node

This example shows how four different coexisting indices are defined in parallel, namely a simple text index, a simple standard index and two different compound indices:

```
<xs:element ...>
  ...
  <tsd:elementInfo>
    <tsd:physical>
     [<tsd:which>...</tsd:which>]
      <tsd:native>
        <tsd:index>
          <tsd:text/>
          <tsd:standard/>
          <tsd:standard>
```

```
            <tsd:field xpath="C"/>
            <tsd:field xpath="B/@b" />
          </tsd:standard >
          <tsd:standard>
            <tsd:field xpath="D"/>
            <tsd:field xpath="B/@b" />
          </tsd:standard >
          ...
        </tsd:index>
      </tsd:native>
    </tsd:physical>
  </tsd:elementInfo>
</xs:element>
```

Example 6: Defining a Reference Index

This example schema provides 3 levels of references:

1. the document root;
2. `A` appearing with multiplicity greater than 1;
3. `B` appearing with multiplicity greater than 1.

Below B there are two nodes `C` and `D` which have indices (one of type `text` and one type `standard` with datatype `integer`), which refer to `B`. On `B` a reference index is declared, which points to `A` and `A` refers to the document root.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:tsd="http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:annotation>
    <xs:appinfo>
      <tsd:schemaInfo name="DocTreeIndex">
        <tsd:collection name="DocTreeIndex"></tsd:collection>
        <tsd:doctype name="Doc" />
        <tsd:doctype name="Doc1" />
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name="Doc">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="A" maxOccurs="unbounded">
          <xs:annotation>
            <xs:appinfo>
              <tsd:elementInfo>
                <tsd:physical>
```

```
                <tsd:native>
                  <tsd:index>
                    <tsd:reference />
                  </tsd:index>
                </tsd:native>
              </tsd:physical>
            </tsd:elementInfo>
          </xs:appinfo>
        </xs:annotation>
        <xs:complexType>
          <xs:sequence>
            <xs:element name="B" maxOccurs="unbounded">
              <xs:annotation>
                <xs:appinfo>
                  <tsd:elementInfo>
                    <tsd:physical>
                      <tsd:native>
                        <tsd:index>
                          <tsd:reference>
                            <tsd:refers>/Doc/A</tsd:refers>
                          </tsd:reference>
                        </tsd:index>
                      </tsd:native>
                    </tsd:physical>
                  </tsd:elementInfo>
                </xs:appinfo>
              </xs:annotation>
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="C" type="xs:string">
                    <xs:annotation>
                      <xs:appinfo>
                        <tsd:elementInfo>
                          <tsd:physical>
                            <tsd:native>
                              <tsd:index>
                                <tsd:text>
                                  <tsd:refers>/Doc/A/B</tsd:refers>
                                </tsd:text>
                              </tsd:index>
                            </tsd:native>
                          </tsd:physical>
                        </tsd:elementInfo>
                      </xs:appinfo>
                    </xs:annotation>
                  </xs:element>
                  <xs:element name="D" type="xs:integer">
                    <xs:annotation>
                      <xs:appinfo>
                        <tsd:elementInfo>
                          <tsd:physical>
                            <tsd:native>
```

```
                              <tsd:index>
                                <tsd:standard>
                                  <tsd:refers>/Doc/A/B</tsd:refers>
                                </tsd:standard>
                              </tsd:index>
                            </tsd:native>
                          </tsd:physical>
                        </tsd:elementInfo>
                      </xs:appinfo>
                    </xs:annotation>
                  </xs:element>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="refDoc1" type="xs:string"></xs:element>
    </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="Doc1">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="A1" type="xs:string">
        <xs:annotation>
          <xs:appinfo>
            <tsd:elementInfo>
              <tsd:physical>
                <tsd:native>
                  <tsd:index>
                    <tsd:standard></tsd:standard>
                  </tsd:index>
                </tsd:native>
              </tsd:physical>
            </tsd:elementInfo>
          </xs:appinfo>
        </xs:annotation>
      </xs:element>
      <xs:element name="F" type="xs:string"></xs:element>
    </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Example 7: Defining a Computed Index

This example illustrates three different types or levels of usage of computed indexes:

basic usage;
a pseudo-compound index concatenating two strings with a delimiter;
for sorting.

As described in the *Performance Guide*, section Advanced Indexes, a computed index and its usage consist of several components as given below:

- XQuery Module
- Schema
- XQuery
- Data

XQuery Module

```
module namespace demo="http://example.computedIndex.org"

declare function demo:getTitleLowerCase($in as node()) as xs:string*
{
 for $t in $in//title
 return lower-case(string($t))
}

declare function demo:getAuthorFullName($in as node()) as xs:string*
{
 for $a in $in//author
 return string-join( ( string($a/last), string($a/first) ), "," )
}

declare function demo:getAuthorsTotalNameLength($in as node()) as xs:integer
{
   xs:integer(sum(
   for $a in $in//author
   return string-length(string($a/last))
 ))
}
```

Schema

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema" xmlns:tsd = ↩
"http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
  <xs:annotation>
    <xs:appinfo>
      <tsd:schemaInfo name = "books" ↩
xmlns:tsd="http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
        <tsd:collection name = "mybib"/>
        <tsd:doctype name = "bib">
          <tsd:physical>
            <tsd:index>
              <tsd:standard>
                <tsd:computed name='compIndex-1' function='p:getTitleLowerCase'
     type='xs:string' xmlns:p='http://example.computedIndex.org'/>
              </tsd:standard>
              <tsd:standard>
                <tsd:computed name='compIndex-2' function='p:getAuthorFullName'
     type='xs:string' xmlns:p='http://example.computedIndex.org'/>
              </tsd:standard>
              <tsd:standard>
                <tsd:computed name='compIndex-3' ↩
function='p:getAuthorsTotalNameLength'
     type='xs:int' xmlns:p='http://example.computedIndex.org'/>
              </tsd:standard>
            </tsd:index>
          </tsd:physical>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name = "bib">
    <xs:complexType>
      <xs:sequence>
      <xs:choice>
        <xs:element ref = "book"></xs:element>
        <xs:element ref = "article"></xs:element>
      </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
   <xs:element name = "article">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref = "title"></xs:element>
        <xs:choice>
          <xs:element ref = "author" maxOccurs = "unbounded"></xs:element>
          <xs:element ref = "editor" maxOccurs = "unbounded"></xs:element>
        </xs:choice>
      </xs:sequence>
```

```
      <xs:attribute name = "year" type = "xs:integer" use = "required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name = "book">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref = "title"></xs:element>
        <xs:choice>
          <xs:element ref = "author" maxOccurs = "unbounded"></xs:element>
          <xs:element ref = "editor" maxOccurs = "unbounded"></xs:element>
        </xs:choice>
        <xs:element ref = "publisher"></xs:element>
        <xs:element ref = "price"></xs:element>
      </xs:sequence>
      <xs:attribute name = "year" type = "xs:integer" use = "required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name = "author">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref = "last"></xs:element>
        <xs:element ref = "first"></xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name = "editor">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref = "last"></xs:element>
        <xs:element ref = "first"></xs:element>
        <xs:element ref = "affiliation"></xs:element>
      </xs:sequence>
      <xs:attribute name = "year" type = "xs:integer" use = "required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name = "publisher">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base = "xs:string">
          <xs:attribute name = "year" type = "xs:string" use = "required"/>
        </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  </xs:element>
  <xs:element name = "title" type = "xs:string"/>
  <xs:element name = "last" type = "xs:string"/>
  <xs:element name = "first" type = "xs:string"/>
  <xs:element name = "affiliation" type = "xs:string"/>
  <xs:element name = "price" type = "xs:string"/>
</xs:schema>
```

XQuery

The following XQuery statements are evaluated using the computed indexes defined above:

```
import module namespace demo='http://example.computedIndex.org';
for $x in collection('mybib')/bib
where demo:getTitleLowerCase(root($x))='tcp/ip illustrated'
return $x

import module namespace demo='http://example.computedIndex.org';
for $x in collection('mybib')/bib
where demo:getAuthorFullName(root($x)) = 'Stevens,W.'
return $x

import module namespace demo='http://example.computedIndex.org';
for $x in collection('mybib')/bib
order by demo:getAuthorsTotalNameLength(root($x))"
return $x
```

Note that all of these XQuery statements using the respective XQuery functions receive the document root as input parameter. This is required in order to enable the XQuery processor to take advantage of using the computed index.

Data

The following data can be used to test the computed indexes as defined above:

```
<bib>
    <book year="1994">
        <title>TCP/IP Illustrated</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher year="1990">Addison-Wesley</publisher>
        <price>65.95</price>
    </book>
</bib>
<bib>
    <book year="1992">
        <title>Advanced Programming in the Unix environment</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher year="2000">Addison-Wesley</publisher>
        <price>65.95</price>
    </book>
</bib>
<bib>
    <book year="2000">
        <title>Data on the Web</title>
        <author><last>Abiteboul</last><first>Serge</first></author>
        <author><last>Buneman</last><first>Peter</first></author>
        <author><last>Suciu</last><first>Dan</first></author>
        <publisher year="2001">Morgan Kaufmann Publishers</publisher>
```

```
        <price>39.95</price>
    </book>
</bib>
<bib>
    <book year="1999">
        <title>The Economics of Technology and Content for Digital TV</title>
        <editor year="1973">
               <last>Gerbarg</last><first>Darcy</first>
               <affiliation>CITI</affiliation>
        </editor>
        <publisher year="2002">Kluwer Academic Publishers</publisher>
        <price>129.95</price>
    </book>
</bib>
<bib>
    <book year="1985">
        <title>Computer-Aided Database Design: the DATAID approach</title>
        <author><last>Di Leva</last><first>Antonio</first></author>
        <author><last>Albano</last><first>Antonio</first></author>
        <publisher year="1985">North-Holland</publisher>
        <price>12.95</price>
    </book>
</bib>
<bib>
    <article year="1999">
        <title>The Economics of Technology and Content for Digital TV</title>
        <editor year="1973">
               <last>Gerbarg</last><first>Darcy</first>
               <affiliation>CITI</affiliation>
        </editor>
    </article>
</bib>
```

Index

M

N

O

P

S

T

U

W

X