Tamino

Performance Guide

Version 9.7

April 2015

WEBMETHODS

This document applies to Tamino Version 9.7.

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

Document ID: INS-PERFORMANCE-97-20160318

Table of Contents

Performance Guide

Performance as discussed in this document refers to the speed with which Tamino can index, store, and retrieve data. It is influenced by many different factors, such as your overall working environment, including hardware configuration and web access, the way your Tamino database is set up and tuned, and how your data is mapped and indexed. Also, your schema definition plays a decisive role. This guide provides information about how you can optimize Tamino performance with regard to the factors mentioned above. The following aspects are covered:

Use of Database Parameters	Lists parameters and shows how to configure them to increase overall performance.
Data Modeling	Describes how to design an efficient data model.
Tuning Schemas and Queries	Shows principles for efficient schemas and queries.
Advanced Indexes	Describes additional possibilities for indexing data.
Efficient Queries: XQuery	Describes how to design efficient queries with XQuery.
Efficient Queries: X-Query	Describes how to design efficient queries and very fast queries with X-Query
Query Processing Analysis	Shows how to retrieve information about the execution plan of a query.
Suppressed Lookup of Index Entries	Shows how to create load lists for quicker indexing.
Hardware Configuration	Gives advice about an efficient hard- and software configuration for Tamino.
Performance Tuning - A Case Study	Describes a practical example for performance tuning.

1 Use of Database Parameters

Tamino has about 30 database parameters, some of which influence overall performance. In the following table, you will find a short description of the parameters that influence performance and recommendations about how to set them. For a general description of database parameters, see Tamino Manager, Creating a Database.

Name	Description/Performance
Server Properties	
Buffer pool size	This is the size of the buffer pool of the Tamino Server. The higher this value is set, the shorter the I/O waiting times. The maximum should not exceed the size of physical memory in order to avoid swapping (keep other memory consuming resources in mind!). If Tamino starts receiving regular page faults, the value is probably too high compared to the size of the physical memory. **Tip:** In general, enlarging the buffer pool will lead to less I/Os and thus to better performance, as long as the system is not swapping and as long as the buffer pool is fully used (this can be checked using the Database Activity display of System Management Hub). A good practice is increasing the buffer pool by a constant value (for example 200 MB) and to simultaneously check the buffer pool hit rate within the Database Activity display. As long as the hit rate increases, performance improvements are to be expected. As soon as the buffer pool hit rate does not increase any longer when increasing the buffer pool size the optimum setting has been surpassed. Obviously the buffer pool can't be exceeded beyond the memory available.
Dynamic pool size	This is the size of the internal dynamic working pool of the Tamino server. It is used to keep result sets and for sorting. If this parameter is set too low, information will be written to temporary working space which leads to disk I/O.
Request log file maximum size	XML request logging may degrade performance to a small extent. Hence, performance data gained from XML request logging should be handled with care.
XML Properties	
XML work threads (XWT)	This is the number of threads used to process XML commands, which can be processed in parallel. If no XWT is available (too many parallel requests), the request will be kept in a queue. The thread is busy until the result is sent or an error occurs. If the value is

Name	Description/Performance
	too high, the threads may impede each other. On the other hand, if the value is too low, applications doing updates may become too slow due to locking.
Word fragment index	This specifies whether Tamino should create a word fragment index when the search type is set for text indexing. This accelerates the resolution of queries involving text strings such as `name~="*string*"`, which would find name nodes whose values contain "string". Specifying "yes" for the word fragment index thus enhances the performance of queries, but negatively impacts performance and space consumption when loading and indexing data. The index can be switched on and off with the Tamino Manager. The change is effective from the next start of the Tamino server. Note that switching on this parameter for a database that already contains data will result in very long startup times for the first server start after the enabling of the index. If you switch the word fragment index on or off, we recommend that you shut down and start up the database and then take a backup before you make any content changes to the database.
XML maximum request duration	The maximum duration of each request is limited to the specified number of seconds. If a request takes longer than the limit that is specified with this parameter, the request will be aborted, and an error message is returned.
XQuery document cache size	This parameter defines the capacity of the document cache that is used for XQuery processing for each request. Document caching improves the performance of certain queries like join queries or sort-by queries where the sorting is not done via an index. Also certain XQuery update statements can benefit from a big document cache. The capacity of the XQuery document cache is specified in MB. The default capacity is 20 MB. For applications with a high parallel query load it may be necessary to reduce the parameter.

2 Data Modeling

The first step to efficiently use Tamino is to get the data model right. The following aspects should be considered:

- The document types should implement whole business objects and documents. You should avoid "relational" designs such as first Normal Form. Business objects represented by an ensemble of flat tables are not suitable for native XML databases. In some cases, however, it may be necessary to split a large business object into several documents.
- Avoid large documents. Large documents can slow down processing considerably. For example the current Tamino version compresses documents larger than 32 KB to save disk space and to speed up disk access (see also chapter ***Result Size***). However, it is possible to "reassemble" many small documents back together into one large document by using joins and projections in XQuery.

  Also, when a single document contains several business objects, you may run into performance problems due to locking conflicts. Because locks are set on the document level, you would lock all business objects contained in the updated document, even if a certain business object is not affected by the update. This will prohibit other users from concurrently accessing or updating these business objects, depending on the isolation level.
- If a document contains clearly identifiable "hot spots" and "cold areas", i.e. a small area is accessed frequently while another large area is accessed only rarely, consider separating these two areas into two documents. This will increase the processing speed for the frequently accessed area.
- Sometimes it can be appropriate to re-introduce redundant data elements in order to speed up retrieval. The downside to this is that updating becomes more complicated and takes more time.

3 Tuning Schemas and Queries

You can influence the performance of Tamino by adhering to the following general principles:

- Before defining indexing information for your Tamino schema, you should as far as possible anticipate what type of query is most likely to be run against which nodes. These nodes are the candidates for index information.
- In deciding whether to index a node with a `text` or a `standard` index, remember that a text index provides optimal performance only when text search operations will be used in queries.
- Consider using a word fragment index only in those cases where it is absolutely necessary - by default, this option is set to **no** because using the word fragment index causes substantial overhead (all possible word fragments must be extracted from words and stored as index values). A word fragment index is only useful in combination with a text index. For details, see the chapter on Indexing in the Advanced Concepts documentation.
- Using unreasonable queries can be detrimental to performance. The more specific or direct the query, the faster the response.
- Consider whether your intended usage scenario would benefit from multipath indexes, compound indexes and reference indexes.

The success and speed of a query run against the Tamino database depends on the combination of index type defined for an XML object and the type of query that will be run against it. Another factor is the mapping type that determines where the object is stored. If typical requests of the application are known, the index types can then be set to support the most typical queries.

This section examines these factors and indicates the settings that are most likely to bring you optimal performance.

Using Structure Index

The default value "CONDENSED" on the structure-index attribute on the Doctype specifies that instance nodes not explicitly mapped in the schema will be registered in the structure index. This provides index support for wildcard and descendant operators in structural queries for nodes not mapped in the schema and thus improves query performance in those cases, in which there are no resulting documents. If you use for example the query "_XQL=a/b" with a condensed index, the result shows that the path a/b does not exist in the respective doctype. Thus query performance increases. With a condensed index, Tamino's optimizer knows if the index information is complete. Queries with wildcards (*) or the path operator (//) are much faster.

A full description of general attributes is given in *Tamino XML Schema Reference Guide*.

An enhanced query performance can be achieved with the use of the value "FULL" for the structure index. In this case, the structure index shows which path occurs in which document. The value "FULL" causes all instance nodes not defined in the schema to be registered by their structure and therefore significantly improves the performance of queries for such nodes. Another situation, where the "FULL" structure index is very useful is the case where optional elements in the schema

are used in queries and there are only a few document with this optional element in the database. The price for using this setting is a slower load operation and a larger index. Though the default value is recommended for most applications, the value "FULL" has its value in situations in which heterogeneous instances of a doctype are expected with a number of element names that may not be known at schema definition time. As with other mapping aspects that impact performance, it is a trade-off between loading time and index size on the one hand and expected nature of the documents and queries on the other.

The value "No" is also recommended for documents with known elements in a random structure (for example, XHTML).

Basic Indexing

Text Index

The text index creates an index for full text search capabilities. This index type is optimal for retrieval of words, as it supports the contains operator (see `tf:ContainsText`) , as well as the operators "adj" (see `tf:ContainsAdjacentText)` and "near" (see `tf:ContainsNearText`). Examples of using this operator are:

```
_xql=/patient[name/surname~='atkin*']
```

```
_xquery=declare namespace ↩
tf="http://namespaces.softwareag.com/tamino/TaminoFunction
for $p in input()/patient
where tf:containsText (/$p/name/surname, "atkin*")
return $p
```

One of the effects of the contains operator is that the request is normalized, basically meaning that the request is not case-sensitive (however, this behavior depends on the settings of the schema element `ino:transliteration`; for a detailed description see section Representation and Handling of Characters in *Unicode and Text Retrieval*). The effect of the wildcard character (*) is that all instances of "patient" are found whose surnames contain a word starting with "atkin".

Note that a text index creates an index for the whole content of the node. Node content in this sense means the concatenation of all descendant nodes (but not attribute nodes) that contain text. Setting a text index on intermediate nodes should therefore be practiced with caution.

The impact of a text index on performance is that creating index data takes time, which is an important consideration when loading data. The more data is loaded, the greater the impact on load time. You must therefore use it with a degree of caution rather than liberally.

Text indexing is not possible for data stored into doctypes with the `noConversion` flag set.

Standard Index

Define all primary and foreign keys used in the conceptual model as index of type standard.

Nodes that are used as sort criteria should be defined as indices of type standard also.

With a standard index, if an element has a string data type of `xs:string`, a string index is created. For this element, only string comparisons can be handled via the index, making the internal lookup much faster, and numeric comparisons are done without an index lookup. In the case of the "=" operator, the index is used for a preselection. Similarly, if the element has a numeric data type of `xs:integer` or `xs:float`, only numeric comparisons can be handled by the index. String comparisons must be done without an index lookup. Thus, if an element `born` has a numeric index, then `born > 1950` will generally be answered much faster than `born > "1950"` because this element has a numeric index, but no string index.

Standard and Text Index

For performance reasons, it is recommended to use this type of indexing only if it is absolutely necessary, because it means both a text *and* a standard index is created. This may lead to slower indexing operations.

However, this index type has its uses. An example of this is if you wish to search for patients whose surnames start with “At” (requires text index) but you also want to accelerate sorting the output alphabetically by surname (requires standard index).

Result Size

The size of the query result set should be small, because the following rule applies: The larger the size of the result data, the longer the query response time. Especially on large databases, it is easy to construct queries which deliver large amounts of data. That is a very time-consuming process. There are two reasons for the time consumption:

- Tamino has to construct the result set in memory first.
- Then all the data must be transferred from the server to the client.

If you are not sure if the size of the result data will block the system, it is recommended to compute the expected size of the query result set. Use the function `count()`, or a cursor, or formulate queries which will request the `ino:id` of the XML instances instead of the whole data (e.g.: `/my_doctype[ac~="willi"]/@ino:id`). In the following query, use the ino:id to retrieve the data.

Alternatively, you can use cursoring to restrict the result set to a subset of the documents that match the query. For more information, see for example the section The cursor command in the *X-Machine Programming* documentation.

4 Advanced Indexes

In addition to the well-known standard and text indexes, Tamino offers the following advanced indexes:

- unique keys
- multipath indexes
- computed indexes
- compound indexes
- reference indexes

The impact on performance of these indexes is discussed in this chapter. You should be familiar with the syntax and concepts of these indexes as described in the *Tamino XML Schema Reference Guide* and the *Tamino XML Schema User Guide*. The information is organized under the following topics:

General Considerations

The purpose of indexes is to improve query performance. However, this is done at the disadvantage of a higher disk space consumption and a higher effort when documents are inserted, modified or deleted. Thus it should be thoroughly considered if it is really necessary to create an index (which means there are enough queries that can benefit from the index) and whether the disadvantages can be tolerated.

Unique Keys

From a logical point of view, a unique key is just an assertion: Tamino guarantees that each value of a unique key appears only once within the doctype. Internally, Tamino uses an index for each unique key in order to easily keep track of the already existing values. In addition to their main task of duplicate detection, these indexes are also used during query evaluation. Hence, unique keys can improve query performance.

If a unique key is defined with one component field, a standard index will be created for that field. If there are several fields, a compound index will be created at the root element. The example below shows a schema with unique key definitions, followed by a schema that shows the indexes created by Tamino (note that the original schema is not modified by Tamino, the second schema is just shown to illustrate the index creation). Hence, from a performance point of view, a unique key behaves either like a standard index or like a compound index.

Schema with unique key definitions:

```
<?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 = "unique">
        <tsd:collection name = "MyCollection"></tsd:collection>
        <tsd:doctype name = "A">
          <tsd:logical>
            <tsd:content>closed</tsd:content>
            <tsd:unique name = "simple-key">
              <tsd:field xpath = "D"></tsd:field>
            </tsd:unique>
            <tsd:unique name = "compound-key">
              <tsd:field xpath = "B/@b"></tsd:field>
              <tsd:field xpath = "C"></tsd:field>
            </tsd:unique>
          </tsd:logical>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name = "A">
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "B">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension>
                <xs:attribute name = "b" type = "xs:string" use = "required">
               </xs:attribute>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:element name = "C" type = "xs:string"></xs:element>
        <xs:element name = "D" type = "xs:string"></xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Corresponding schema with indexes created by Tamino:

```
<?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 = "unique">
        <tsd:collection name = "MyCollection"></tsd:collection>
        <tsd:doctype name = "A">
          <tsd:logical>
            <tsd:content>closed</tsd:content>
            <tsd:unique name = "simple-key">
              <tsd:field xpath = "D"></tsd:field>
            </tsd:unique>
            <tsd:unique name = "compound-key">
              <tsd:field xpath = "B/@b"></tsd:field>
              <tsd:field xpath = "C"></tsd:field>
            </tsd:unique>
          </tsd:logical>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name = "A">
    <xs:annotation>
      <xs:appinfo>
        <tsd:elementInfo>
          <tsd:physical>
            <tsd:native>
              <tsd:index>
                <tsd:standard>
                  <tsd:field xpath = "B/@b"></tsd:field>
                  <tsd:field xpath = "C"></tsd:field>
                </tsd:standard>
              </tsd:index>
            </tsd:native>
          </tsd:physical>
        </tsd:elementInfo>
      </xs:appinfo>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "B">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base = "xs:string">
                <xs:attribute name = "b" type = "xs:string" use = "required">
                 </xs:attribute>
              </xs:extension>
```

```
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:element name = "C" type = "xs:string"></xs:element>
        <xs:element name = "D" 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:schema>
```

Note: Although Tamino automatically creates indexes in order to implement unique key constraints, it is recommended to explicitly define the corresponding index in the schema if you rely on the performance improvement. Tamino will detect when an index definition matches a unique key constraint, and only one index will be created. The benefit is that such an explicitly defined index will survive if the unique key constraint is modified or removed.

Multipath Index

A multipath index is an index that covers several paths: if each of those paths had its own index, the corresponding multipath index can be seen as the union of those indexes. As a feature, multipath is an add-on option for other indexes. It can be used with standard, compound, and text indexes. See the respective section in the *Tamino XML Schema Reference Guide* for detailed rules about creating a multipath index.

The multipath feature supports queries in the following scenarios:

- Highly-connected structures: Global elements or attributes with index are referenced from many places in the schema (which might become a problem as the number of distinct indexes is limited).
- Recursive structures: Each occurrence of an element or attribute in a recursive structure is to be indexed.

- Arbitrary path sets: Arbitrary path sets can be combined into one multipath index, if the rules apply (paths have to have the same type of index and the same data types).

The following examples illustrate these scenarios.

Highly-connected Structures

This example schema has several types of chapters, each of which has a title which is defined in a global element. The title has a text index, and instead of defining a separate index for each possible path, one common multipath index is defined which is used for any possible path to the `Title` element.

Example: Highly-connected Schema

```
<?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 = "highly-connected">
        <tsd:collection name = "MyCollection"></tsd:collection>
        <tsd:doctype name = "Document">
          <tsd:logical>
            <tsd:content>closed</tsd:content>
          </tsd:logical>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name = "Title" type = "xs:string">
    <xs:annotation>
      <xs:appinfo>
        <tsd:elementInfo>
          <tsd:physical>
            <tsd:native>
              <tsd:index>
                <tsd:text>
                  <tsd:multiPath>allTitlesIndex</tsd:multiPath>
                </tsd:text>
              </tsd:index>
            </tsd:native>
          </tsd:physical>
        </tsd:elementInfo>
      </xs:appinfo>
    </xs:annotation>
  </xs:element>
  <xs:element name = "Document">
    <xs:complexType>
      <xs:sequence>
```

```
        <xs:element name = "Chapter1">
          <xs:complexType>
            <xs:sequence>
              <xs:element ref = "Title"></xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name = "Chapter2">
          <xs:complexType>
            <xs:sequence>
              <xs:element ref = "Title"></xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name = "Chapter3">
          <xs:complexType>
            <xs:sequence>
              <xs:element ref = "Title"></xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

This multipath index is used in an optimal way by queries like the following (the first example query uses XQuery syntax, followed by the same example in X-Query syntax respectively):

```
declare namespace tf="http://namespaces.softwareag.com/tamino/TaminoFunction"
for $d in input()/Document
where tf:containsText ($d//Title, "some text")
return $d
```

```
_XQL = /Document[.//Title ~= "some text"]
```

It finds all documents where an arbitrary title, regardless of its path, fulfils the search criterion. The result is found by performing one index lookup. Without the multipath index, there has to be a separate index for each path, and the result of several index lookups had to be combined by an OR operation.

The next example evaluates the criterion against one particular path:

```
declare namespace tf="http://namespaces.softwareag.com/tamino/TaminoFunction"
for $d in input()/Document
where tf:containsText ($d/Chapter1/Title, "some text")
return $d
```

```
_XQL = /Document[Chapter1/Title ~= "some text"]
```

This query also makes use of the multipath index. But as the index has no knowledge about the path in which a particular value occurs, the index can only deliver a superset of the real result. From the viewpoint of the index, the criterion could be fulfilled by `Chapter1` or `Chapter2` or `Chapter3`. This superset has to be filtered by post-processing.

Recursive Structures

This example schema defines a chapter that has a title, and that contains a nested chapter. The title has a text index. Without a multipath index, there is no chance to index every possible nesting level. Using `tsd:which`, only a finite number of nesting levels can be explicitly indexed.

Recursive Schema

```
<?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 = "recursive">
        <tsd:collection name = "MyCollection"></tsd:collection>
        <tsd:doctype name = "Document">
          <tsd:logical>
            <tsd:content>closed</tsd:content>
          </tsd:logical>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name = "Document">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref = "Chapter"></xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name = "Chapter">
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "Title" type = "xs:string">
          <xs:annotation>
            <xs:appinfo>
              <tsd:elementInfo>
```

```
                <tsd:physical>
                  <tsd:native>
                    <tsd:index>
                      <tsd:text>
                        <tsd:multiPath>nestedTitlesIndex</tsd:multiPath>
                      </tsd:text>
                    </tsd:index>
                  </tsd:native>
                </tsd:physical>
              </tsd:elementInfo>
            </xs:appinfo>
          </xs:annotation>
        </xs:element>
        <xs:element ref = "Chapter" minOccurs = "0"></xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The queries supported by this multipath index are very similar to the highly-connected scenario.

```
declare namespace tf="http://namespaces.softwareag.com/tamino/TaminoFunction"
for $d in input()/Document
where tf:containsText ($d//Title, "some text")
return $d
```

```
_XQL = /Document[.//Title ~= "some text"]
```

This query finds all documents where an arbitrary title, regardless of its nesting level, fulfils the search criterion. The result is found by performing one index lookup. Without the multipath feature, this query can only be supported by indexes if every actually occurring nesting level of `Title` is explicitly indexed by a `tsd:which` statement.

The next example evaluates the criterion against one particular nesting level:

```
declare namespace tf="http://namespaces.softwareag.com/tamino/TaminoFunction"
for $d in input()/Document
where tf:containsText ($d/Chapter/Chapter/Title, "some text")
return $d
```

```
_XQL = /Document[Chapter/Chapter/Title ~= "some text"]
```

This query also makes use of the multipath index. But as the index has no knowledge about the nesting level at which a particular value occurs, the index can only deliver a superset of the real result: from the viewpoint of the index, the criterion could be fulfilled by `Chapter/Title` or `Chapter/Chapter/Title`, and so on. This superset has to be filtered by post-processing.

Arbitrary Path Sets

The previous examples are based on the use of global elements (which is of course mandatory for recursion). The multipath feature, however, is not restricted to global elements. The following example shows a document that has an introduction with a subtitle, and two chapters with a title (where each title is modeled locally under its parent). Each of these three title definitions has its own multipath definition. As these definitions specify the same multipath label, the schema actually defines one multipath index, with three participating paths.

Example: Arbitrary Path Set

```
<?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 = "path-set">
        <tsd:collection name = "MyCollection"></tsd:collection>
        <tsd:doctype name = "Document">
          <tsd:logical>
            <tsd:content>closed</tsd:content>
          </tsd:logical>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name = "Document">
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "Introduction">
          <xs:complexType>
            <xs:sequence>
              <xs:element name = "Subtitle" type = "xs:string">
                <xs:annotation>
                  <xs:appinfo>
                    <tsd:elementInfo>
                      <tsd:physical>
                        <tsd:native>
                          <tsd:index>
                            <tsd:text>
                              <tsd:multiPath>allTitles</tsd:multiPath>
                            </tsd:text>
                          </tsd:index>
                        </tsd:native>
                      </tsd:physical>
                    </tsd:elementInfo>
                  </xs:appinfo>
                </xs:annotation>
              </xs:element>
```

```
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name = "Chapter1">
        <xs:complexType>
          <xs:sequence>
            <xs:element name = "Title" type = "xs:string">
              <xs:annotation>
                <xs:appinfo>
                  <tsd:elementInfo>
                    <tsd:physical>
                      <tsd:native>
                        <tsd:index>
                          <tsd:text>
                            <tsd:multiPath>allTitles
                            </tsd:multiPath>
                          </tsd:text>
                        </tsd:index>
                      </tsd:native>
                    </tsd:physical>
                  </tsd:elementInfo>
                </xs:appinfo>
              </xs:annotation>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name = "Chapter2">
        <xs:complexType>
          <xs:sequence>
            <xs:element name = "Title" type = "xs:string">
              <xs:annotation>
                <xs:appinfo>
                  <tsd:elementInfo>
                    <tsd:physical>
                      <tsd:native>
                        <tsd:index>
                          <tsd:text>
                            <tsd:multiPath>allTitles
                           </tsd:multiPath>
                          </tsd:text>
                        </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:schema>
```

Queries similar to the following examples make use of the multipath index:

```
declare namespace tf="http://namespaces.softwareag.com/tamino/TaminoFunction"
for $d in input()/Document
where    tf:containsText ($d/Introduction/Subtitle, "some text")
      or tf:containsText ($d//Title, "some other text")
return $d
```

```
_XQL = /Document[Introduction/Subtitle ~= "some text"
                 or .//Title ~= "some other text"]
```

```
declare namespace tf="http://namespaces.softwareag.com/tamino/TaminoFunction"
for $d in input()/Document
where    tf:containsText ($d/Introduction/Subtitle, "some text")
return $d
```

```
_XQL = /Document[Introduction/Subtitle ~= "some text"]
```

```
declare namespace tf="http://namespaces.softwareag.com/tamino/TaminoFunction"
for $d in input()/Document
where    tf:containsText ($d/Chapter1/Title, "some text")
return $d
```

```
_XQL = /Document[Chapter1/Title ~= "some text"]
```

```
declare namespace tf="http://namespaces.softwareag.com/tamino/TaminoFunction"
for $d in input()/Document
where     tf:containsText ($d/Chapter1/Title, "some text")
      and tf:containsText ($d/Chapter2/Title, "some other text")
return $d
```

```
_XQL = /Document[Chapter1/Title ~= "some text"
                 and Chapter2/Title ~= "some other text"]
```

In all these cases, post-processing is required to filter the result of the index scan. The reason is again that the index has no knowledge about the path in which a particular value occurs.

Computed Index

A computed index is even more powerful than a multipath indexes, with the current restriction that a computed index may be neither a text index nor a compound index. Instead of adding the index definition to all nodes (or paths) to be included in a multipath index, the computed index refers to an XQuery function which is defined in a module stored in Tamino via the QName of the XQuery function. This XQuery function may compute one or more index entries based on arbitrary nodes and their values in the XML document being stored in a doctype.

A computed index consists of:

- an XQuery module defining the indexing function(s)
- the schema defining the computed indexes referring to the indexing functions
- an XQuery query taking advantage of the computed index by using the indexing function, for which the root node of each document will passed as an argument. The indexing function must be used either in a comparison or in an "order by" clause.

An indexing function must have the following signature:

- Exactly one parameter of type "node()";
- The return type is the QName of a known simple type; at the moment it must be a type predefined by XML Schema. Hence, a QName such as "xs:integer" might be specified, with an additional occurrence indicator such as "?" or "*". A return types such as "node()" or "item()" with an optional occurrence indicator is not acceptable.

The `type` attribute of `tsd:computed`, which is typically the same as the declared return type of the indexing function, must specify a simple type that is predefined in XML Schema.

For examples and additional aspects, please refer to the following documentation sections:

- XML Schema User Guide > Appendix 5: Example Schemas for Indexing
- XQuery User Guide > Advanced Usage > Defining and Using Modules
- X-Machine Programming > Maintaining Tamino Indexes
- Machine Programming > Requests using X-Machine Commands > _admin

Compound Index

A compound index combines values from different component fields into one index value. The following schema has a `Name` element with `Firstname`, `Initial`, and `Lastname` children. There is a compound index located at the `Name` element, having `Firstname`, `Initial`, and `Lastname` as components (in that sequence).

Example: Compound Index

```
<?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 = "compound">
        <tsd:collection name = "MyCollection"></tsd:collection>
        <tsd:doctype name = "Document">
          <tsd:logical>
            <tsd:content>closed</tsd:content>
          </tsd:logical>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name = "Document">
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "Name" maxOccurs = "unbounded">
          <xs:annotation>
            <xs:appinfo>
              <tsd:elementInfo>
                <tsd:physical>
                  <tsd:native>
                    <tsd:index>
                      <tsd:standard>
                        <tsd:field xpath = "Firstname"></tsd:field>
                        <tsd:field xpath = "Initial"></tsd:field>
                        <tsd:field xpath = "Lastname"></tsd:field>
                      </tsd:standard>
                    </tsd:index>
                  </tsd:native>
                </tsd:physical>
              </tsd:elementInfo>
            </xs:appinfo>
          </xs:annotation>
          <xs:complexType>
            <xs:sequence>
              <xs:element name = "Firstname" type = "xs:string"></xs:element>
```

```
              <xs:element name = "Initial" type = "xs:string"></xs:element>
              <xs:element name = "Lastname" type = "xs:string"></xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Here are some example documents for this schema:

```
<Document>
  <Name>
    <Firstname>Paul</Firstname>
    <Initial>J</Initial>
    <Lastname>Bloggs</Lastname>
  </Name>
</Document>

<Document>
  <Name>
    <Firstname>Fred</Firstname>
    <Initial>M</Initial>
    <Lastname>Bloggs</Lastname>
  </Name>
  <Name>
    <Firstname>Paul</Firstname>
    <Initial>J</Initial>
    <Lastname>Atkins</Lastname>
  </Name>
</Document>
```

For the first document, the value (Paul,J,Bloggs) is added to the compound index; for the second document, the values (Fred,M,Bloggs) and (Paul,J,Atkins) are added (the tuple notation is used here only for readability purposes, internally Tamino uses a compact serialization format). The following query will make use of the compound index:

```
for $d in input()/Document
for $n in $d/Name
where     $n/Firstname = "Paul"
      and $n/Initial = "J"
      and $n/Lastname = "Bloggs"
return $d
```

```
_XQL = /Document[Name[    Firstname = "Paul"
                      and Initial = "J"
                      and Lastname = "Bloggs"] ]
```

This query finds the first document, the second one does not match because the values `Paul` and `J` appear under one `Name` element, and the value `Bloggs` under another. The query optimizer detects the compound index and scans the index for the value (Paul,J,Bloggs) which is composed from the parts given in the query. Thus, the query can be answered by one index lookup, although it consists of several criteria. Without a compound index, each component had to have its own standard index (in order to have an index-supported query), and several separate index lookups would be necessary.

Moreover, this example shows a much greater performance improvement than only saving index lookups. The criteria are:

- The compound index is hosted by the `Name` element, which means that the compound values are built relative to `Name`,
- and the `Name` element has a multiplicity greater than 1.

In other words, the values of the example compound index are grouped by `Name` elements. Without the compound index, when each component has its own standard index, there is no such grouping, and the index does not know to which occurrence of the `Name` element a particular value belongs. Thus, when executing the given query against three separate indexes, the index lookup will also find the second document (because all requested values appear somewhere in that document), and a subsequent postprocessing step is needed to find the correct result. This unnecessary reading of the second document is avoided with the compound index.

This first query example contains predicates for each component of the compound index. But the compound index can also be used if less predicates appear in the query. The rule is:

- The set of predicates in the query has to refer to the components of the compound index from left to right (in definition sequence).
- The predicates have to be connected by `and`.
- The `and` operation must be in the scope of the location of the compound index (for example, with the compound index on the `Name` element, the `and` must combine paths relative to `Name`).
- The predicates have to be “=” comparisons, with the exception of the last predicate in definition sequence which may be an arbitrary relational comparison operator.

The following query examples illustrate this rule. The first set of queries makes use of the compound index, and postprocessing is not necessary:

```
for $d in input()/Document
for $n in $d/Name
where     $n/Firstname = "Paul"
return $d
```

```
_XQL = /Document[Name[Firstname = "Paul"] ]
```

```
for $d in input()/Document
for $n in $d/Name
where     $n/Firstname > "Paul"
return $d
```

```
_XQL = /Document[Name[Firstname > "Paul"] ]
```

```
for $d in input()/Document
for $n in $d/Name
where     $n/Firstname = "Paul"
      and $n/Initial = "J"
return $d
```

```
_XQL = /Document[Name[    Firstname = "Paul"
                      and Initial = "J"] ]
```

```
for $d in input()/Document
for $n in $d/Name
where     $n/Initial = "J"
      and $n/Firstname = "Paul"
      and $n/Lastname < "Bloggs"
return $d
```

```
_XQL = /Document[Name[    Firstname = "Paul"
                      and Initial = "J"
                      and Lastname < "Bloggs"] ]
```

The next set of queries makes use of the compound index, but an additional postprocessing step is needed because the predicates do not fulfill the rule described above. The query optimizer selects those predicates that fulfill the rule in order to find a minimal superset of the final result using the compound index:

```
for $d in input()/Document
for $n in $d/Name
where     $n/Firstname = "Paul"
      and $n/Lastname = "Bloggs"
return $d
```

```
_XQL = /Document[Name[    Firstname = "Paul"
                      and Lastname = "Bloggs"] ]
```

```
for $d in input()/Document
for $n in $d/Name
where     $n/Firstname = "Paul"
      and $n/Initial > "J"
      and $n/Lastname > "Bloggs"
return $d
```

```
_XQL = /Document[Name[    Firstname = "Paul"
                      and Initial > "J"
                      and Lastname > "Bloggs"] ]
```

The following query cannot use the compound index because there is no predicate for the first component (`Firstname`):

```
for $d in input()/Document
for $n in $d/Name
where     $n/Initial = "J"
      and $n/Lastname = "Bloggs"
return $d
```

```
_XQL = /Document[Name[    Initial = "J"
                      and Lastname = "Bloggs"] ]
```

The following query cannot use the compound index because the `and` operation is not in the scope of the element hosting the compound index (the `Name` element):

```
for $d in input()/Document
where     $d/Name/Firstname = "Paul"
      and $d/Name/Initial = "J"
      and $d/Name/Lastname = "Bloggs"
return $d
```

```
_XQL = /Document[    Name/Firstname = "Paul"
                 and Name/Initial = "J"
                 and Name/Lastname = "Bloggs"]
```

Disk Space Considerations

Compound indexes should be used very carefully if one or even several of the components are multiple (relative to the element hosting the compound index), which means in the example above if a `Name` could consist of several `Firstnames`. In this case, all possible value combinations (the cross-product) are built and added to the index, so that the index can become very large.

Reference Index

A reference index consists of two parts:

- The actual reference index (denoted by `tsd:reference`) is specified at a particular path in the schema. All document occurrences of that path are then assigned a node ID which is unique across the doctype.
- Other indexes (standard, text, compound) located below the reference index can refer to that reference node by specifying `tsd:refers`.

Specifying a reference index makes sense only if

- the reference node has a multiplicity greater than 1,
- and there are at least two referencing indexes.

The schema used for compound indexes (simplified by leaving out the `Initial` element) is now reformulated using a reference index. `Firstname` has a text index, and `Lastname` has a standard index, both referring to the `Name` element:

Example: Reference Index

```
<?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 = "reference">
        <tsd:collection name = "MyCollection"></tsd:collection>
        <tsd:doctype name = "Document">
          <tsd:logical>
            <tsd:content>closed</tsd:content>
          </tsd:logical>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name = "Document">
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "Name" maxOccurs = "unbounded">
          <xs:annotation>
            <xs:appinfo>
              <tsd:elementInfo>
                <tsd:physical>
                  <tsd:native>
                    <tsd:index>
```

```
                    <tsd:reference></tsd:reference>
                  </tsd:index>
                </tsd:native>
              </tsd:physical>
            </tsd:elementInfo>
          </xs:appinfo>
        </xs:annotation>
        <xs:complexType>
          <xs:sequence>
            <xs:element name = "Firstname" type = "xs:string">
              <xs:annotation>
                <xs:appinfo>
                  <tsd:elementInfo>
                    <tsd:physical>
                      <tsd:native>
                        <tsd:index>
                          <tsd:text>
                            <tsd:refers>/Document/Name</tsd:refers>
                          </tsd:text>
                        </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:standard>
                            <tsd:refers>/Document/Name</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:schema>
```

Here are two example documents.

```
<Document>
  <Name>
    <Firstname>Paul</Firstname>
    <Lastname>Bloggs</Lastname>
  </Name>
</Document>

<Document>
  <Name>
    <Firstname>Fred</Firstname>
    <Lastname>Bloggs</Lastname>
  </Name>
  <Name>
    <Firstname>Paul</Firstname>
    <Lastname>Atkins</Lastname>
  </Name>
</Document>
```

When these documents are stored, each `Name` element is assigned a unique ID, and the values for the other indexes are built as usual. The semantic of a referencing index, however, is different: while a classic index contains the information “the value 'Bloggs' appears in the document with ino:id 17”, a reference index says “the value 'Bloggs' appears in the Name node with ID 5”. Thus, a reference index achieves a grouping effect similar to the one described for compound indexes: the values `Fred` and `Bloggs` are grouped under the first `Name` node of the second document, and the values `Paul` and `Atkins` are grouped under the second `Name` node.

Queries can make use of this scenario if

- there are predicates on the referencing index that are combined by an `and`,
- and the `and` operator is in the scope of the reference index.

```
declare namespace tf="http://namespaces.softwareag.com/tamino/TaminoFunction"
for $d in input()/Document
for $n in $d/Name
where     tf:containsText ($n/Firstname, "Paul")
      and $n/Lastname = "Bloggs"
return $d
```

```
_XQL = /Document[Name[    Firstname ~= "Paul"
                      and Lastname = "Bloggs"] ]
```

The index lookups on `Firstname` and `Lastname` and the subsequent intersection find the only `Name` node that fulfills the criteria, postprocessing is avoided. Without a reference index, the index lookup would find both documents (because the values `Paul` and `Bloggs` appear somewhere in both documents), and only postprocessing will find the correct result.

In such a scenario, the query performance is improved significantly because the `and` operation can be performed on the level of the `Name` element instead of the document level. On the `Name` element

level, the intersection delivers already the final result, no document is read from disk only to be rejected by the postprocessor (which would happen without a reference index).

The following example queries make use of the reference index, but there is no performance benefit compared to classic indexes.

```
declare namespace tf="http://namespaces.softwareag.com/tamino/TaminoFunction"
for $d in input()/Document
for $n in $d/Name
where     tf:containsText ($n/Firstname, "Paul")
return $d
```

```
_XQL = /Document[Name[Firstname ~= "Paul"]]
```

This query has only one predicate, thus there is no improvement because there is no intersection on the `Name` element level. Similarly, there would be no improvement if the query had several predicates combined with `or`.

The next query uses an `and` which is not in the scope of the `Name` element. The intersection is on the document level, and the correct result could also be found by classic indexes without postprocessing.

```
for $d in input()/Document
where     $d/Name/Firstname = "Paul"
      and $d/Name/Lastname = "Bloggs"
return $d
```

```
_XQL = /Document[    Name/Firstname = "Paul"
                 and Name/Lastname = "Bloggs"]
```

Actually, the latter examples should be avoided with a reference index. The index lookup of a referencing index (e.g. `Firstname`) delivers node IDs of the reference index (`Name` in this example). These node IDs have to be transformed to document IDs. This is unnecessary overhead if the same result can be achieved by classic indexes. In a “good” reference index scenario, this overhead also exists, but it is by far compensated by saving unnecessary document reads.

Reference Index versus Compound Index

Both reference index and compound index achieve performance improvements in more or less the same scenario where index values can be grouped relative to a particular node that has a multiplicity greater than 1.

Hence the question comes up which one should be preferred if both can be applied. The general recommendation is to use a compound index if it satisfies the query requirements. The reason is that a reference index needs more overhead, as described above.

But a compound index is not always feasible. A reference index is more flexible: It can work with all index types (while a compound index is always a standard index), and it can be nested (there may be several levels with `tsd:reference`).

Selectivity of Compound and Reference Index

As pointed out in the previous chapters, the performance improvement that can be achieved with compound and reference indexes heavily depends on the grouping of values relative to particular nodes (the `tsd:reference` node or the node at which the compound index is defined). The selectivity of a compound or reference index is much higher compared to classic standard indexes if these value groups identify a much smaller result set than without grouping.

In order to determine the selectivity improvement, two different count queries can be issued. The first one counts the number of documents that represents the query result:

```
{-- query based on the compound index example --}

count
(
  for $d in input()/Document
  for $n in $d/Name
  where     $n/Firstname = "Paul"
        and $n/Initial = "J"
        and $n/Lastname = "Bloggs"
  return $d
)
```

```
_XQL = count (/Document[Name   [ Firstname = "Paul"
                 and Initial = "J"
                 and Lastname = "Bloggs"] ] )
```

```
{-- query based on the reference index example --}

declare namespace tf="http://namespaces.softwareag.com/tamino/TaminoFunction"
count
(
  for $d in input()/Document
  for $n in $d/Name
  where     tf:containsText ($n/Firstname, "Paul")
        and $n/Lastname = "Bloggs"
  return $d
)
```

```
_XQL = count(/Document[Name[    Firstname ~= "Paul"
                           and Lastname = "Bloggs"] ])
```

The second one counts the number of documents that had to be read if there was no reference or compound index, and which had then to be presented to the postprocessor:

```
{-- query based on the compound index example --}

count
(
  for $d in input()/Document
  where     $d/Name/Firstname = "Paul"
        and $d/Name/Initial = "J"
        and $d/Name/Lastname = "Bloggs"
  return $d
)
```

```
_XQL = count (/Document[    Name/Firstname = "Paul"
                 and Name/Initial = "J"
                 and Name/Lastname = "Bloggs"] )
```

```
{-- query based on the reference index example --}

declare namespace tf="http://namespaces.softwareag.com/tamino/TaminoFunction"
count
(
  for $d in input()/Document
  where     tf:containsText ($d/Name/Firstname, "Paul")
        and $d/Name/Lastname = "Bloggs"
  return $d
)
```

```
_XQL = count(/Document[    Name/Firstname ~= "Paul"
                       and Name/Lastname = "Bloggs"])
```

If these numbers differ significantly for a representative set of values, this is a good indication to define a compound or a reference index (depending on which one is feasible).

5 Efficient Queries: XQuery

This chapter describes how to increase the performance of Tamino XQuery 4. The following topics are covered:

Using Indexes

XQueries perform much faster if indexes can be used. If an element or attribute is often referred to in a query, it is recommended to define a standard index upon this item in the schema. If this item is moreover used for text retrieval, as, for example, with the `contains()` function, a text index is even more appropriate.

An index - although defined in the schema - might not be used when executing the query. This happens in cases where the connection between the database items to be retrieved and the schema entry upon which the index is defined is not recognized. Reasons for this may be an open-content schema or a path expression in the query that contains element steps, thus passing element definitions for which an `anyType` content model is defined. So for using indexing from within queries, try to keep the areas of uncertain structure very limited, i.e. use close-content schemas and only allow `anyType` where absolutely necessary.

Constructors

Constructors are very helpful to generate XML structures within a query. But used in the wrong way, constructors can degrade the query performance considerably.

Avoid Unnecessary Constructors

First of all, you should avoid constructing XML structures that do not belong to the final query result. The following query gives an example for unnecessary XML construction:

```
for      $a in input()/bib/book
where    for $b in $a/title
         where tf:containsText($b,"XQuery")
         return <true/>
return $a
```

The constructor in the return clause of the nested sub-query creates a `true` element that just indicates the occurrence of a node that satisfies the given search predicates. It is far better to return the matching node:

```
for      $a in input()/bib/book
where    for $b in $a/title
         where tf:containsText($b,"XQuery")
         return $b
return $a
```

Avoid Sorting the Results of an XML Construction

According to the XQuery draft, the result of an XML construction loses all the type information of the parts it has been constructed from. For example, the following query creates a book list where each entry contains a book element with two attributes holding the title and the publishing year:

```
for    $a in input()/bib/book
return <book title="{$a/title}" year="{$a/@year}"/>
```

To sort the result by the publishing year, a sort by clause can be appended to the query. Since the year attribute of the constructed year attribute has lost all of its type information, an `xs:integer()` casting function has to be applied to get the correct ordering:

```
for    $a in input()/bib/book
return <book title="{$a/title}" year="{$a/@year}"/>
sort by(xs:integer(./@year))
```

A more severe consequence of the construction is that the optimizer is not able to perform an index-based sorting. This makes the above query very slow if applied to a big document set. To get around this problem, the sort by clause should be moved to the `for` clause:

```
for $a in input()/bib/book
    sort by(./@year)
return <book title="{$a/title}" year="{$a/@year}"/>
```

Due to the order preserving property of an FLWOR expression in XQuery, the ordering created by the sort by clause is maintained. With the newly introduced order by clause, the query can also be stated like this:

```
for $a in input()/bib/book
    sort by(./@year)
    order by $a/@year
return <book title="{$a/title}" year="{$a/@year}"/>
```

Disjunctive Predicates

Queries with complex search predicates involving “and” and “or” operations are difficult to evaluate efficiently. Most problematic are “or” operations, which may cause index access operations returning big result sets. The number of “or” operations can be reduced if they are applied on comparison predicates that read the same elements or attributes. For example, assuming that we want to get all the entries in a bibliography referencing a book written by Heinrich or Thomas Mann. The query can be stated like this:

```
for    $b in input()/bib/book
where   $b/author[last = "Mann" and (first ="Heinrich" or first ="Thomas")]
return $b
```

The search predicate of the query can be simplified in the following way:

```
for    $b in input()/bib/book
where   $b/author[last = "Mann" and first =("Heinrich","Thomas")]
return $b
```

The query compares the content of the “first” element with the sequence (“Heinrich”,“Thomas”). According to the definition of the general comparison in XQuery, this means that the comparison is successful if the content of the “first” element is either equal to “Heinrich” or “Thomas”. The big advantage of using this kind of predicate is that only index access is performed during evaluation.

Negated Predicates

Tamino 4.4 introduces the index-based evaluation of negated predicates in XQuery expressions. For negating expressions, XQuery provides the `fn:not()` function. For example, to find those books that do not have any title with the value “Data on the Web” the `not()` function can be used as shown in the following query:

```
for    $a in input()/bib/book
   where not($a/title = "Data on the Web")
return $a
```

The following more complex example query retrieves all books except those that were not published in 2000 and written by Dan Suciu:

```
for    $a in input()/bib/book
    where not($a/@year = 2000 and $a/author[first="Dan" and last="Suciu"])
return $a
```

Beside comparison predicates, the `fn:not()` function is also useful in combination with text-retrieval predicates. The following query that retrieves all books that do not have the word “Web” in their title illustrates this:

```
for    $a in input()/bib/book
    where not(tf:containsText($a/title, "Web"))
return $a
```

Also, the absence of elements or attributes can be checked via the not function:

```
for   $a in input()/bib/book
    where not($a/author)
return $a
```

The query finds all books with an empty author list. In summary, index-based evaluation for negated predicates is supported for the following predicate types:

- Comparison predicates on elements and attributes
- Text-retrieval predicates
- Existence checks on elements and attributes

Value Range Predicates

A value range query queries a value range by a conjunctive of two predicates affecting the same node. One predicate specifies a lower bound, and the other specifies an upper bound. For example, the following query finds all books published between 1996 and 2001.

```
for    $a in input()/bib/book
    let $y := $a/@year
    where $y >= 1996 and $y<= 2001
return $a
```

Queries with value range predicates can be optimized by accessing a standard index. Therefore, the value range predicates have to be identified and propagated to a standard index access. For identifying a given conjunction of two comparisons as value range predicate, it has to be verified that both comparisons are acting on the same node. Therefore, the following restrictions hold:

- Node expression must be a variable, variables must be equal for both comparisons.
- If one comparison is a general one, the node delivered by the variable must not be multiple.

As a consequence of the given restrictions, the value range predicate of the example query can be optimized, if the year element is not multiple concerning the book element.

Position Range Predicates

A position range predicate is a predicate qualifier that selects a range of items from a given input sequence. Therefore, it provides a lower and an upper bound for the position of the items that should be part of the result. The following query provides an example for a position range predicate:

```
(input()/bib/book) [position() >= 10 and position() < 20]
```

The query selects those “book” elements with a position greater than or equal to 10 and less than 20. Special range predicates result from implicit lower or upper bounds. The following query selects the first 10 books of the given doctype.

```
(input()/bib/book) [position() <= 10]
```

Here the lower bound is implicitly given. In the next query, the upper bound is implicitly given.

```
(input()/bib/book) [position() >= last() - 10]
```

The query selects the last ten books in the given doctype. The following list gives some examples of range predicates that can be optimized.

- `[position() < last() -10]`
- `[position() > last() - 10]`
- `[position () > 10 and position() < 21]`
- `[position () < 21 and position() > 10]`

Position range predicates are optimized by reducing the number of documents that have to be read during query execution. The prerequisite is that the cardinality of the document set that is read by scanning an index or a doctype is not changed during post-processing. This means that the query must not contain path expressions that deliver more or less than a single item for each document. Another point is that all search predicates must be completely processed using an index access. Assuming that the following assumptions are true:

- Every document of the “bib” doctype holds exactly one `title` element.
- A document of the “bib” doctype may hold more than a single `author` element.
- There is a standard index on the `title` element.

Examples of queries that can be optimized are given in the following list.

- ```
  (input()/bib/book/title) [position() < last() -10]
  ```

- ```
  (for $b in input()/bib/book where $b/title = "Data on the Web" return
        $b) [position() < last() -10]
  ```

- ```
  (for $b in input()/bib/book where $b/author = "Dan Suciu" return $b)
        [position() < last() -10]
  ```

Examples for queries that cannot be optimized are:

- ```
  (input()/bib/book/author) [position() < last() -10]
  ```

- ```
  (let $b := input()/bib/book where $b/title = "Data on the Web" return
        $b) [position() < last() -10]
  ```

Join Ordering

Due to XQuery's ordered data model, the order of "for" clauses in a FLWOR expression is relevant. This means Tamino does not change the order of "for" clauses in a FLWOR expression. Thus you should reorder the "for" clauses manually, if the result order does not matter. A rule of thumb is to start with "for" clauses that iterate expressions that produce small results and that can be evaluated via an index access. For example:

Instead of

```
for    $a in input()/customer,
    $b in input()/vendor
    where   $b/@vno = $a/vno and $b/vname = "Schmidt"
return
            <customers_of_vendor>
            { $b/vname  }
            { $b/@vno  }
            { $a/custname }
            </customers_of_vendor>
```

Use:
```

```
for    $b in input()/vendor
    $a in input()/customer,
    where   $b/@vno = $a/vno and $b/vname = "Schmidt"
return
            <customers_of_vendor>
            { $b/vname  }
            { $b/@vno  }
            { $a/custname }
            </customers_of_vendor>
```

This is because the first variable $a in the first example has to be assigned to all customer documents, whereas the variable $b in the second example has to be assigned only to the vendors with a specific “vname”. This is much more efficient assuming there is a standard index on “vname”.

Index-only Joining

Join queries are evaluated by the Tamino XQuery in an index-based nested-loop manner. This works well if the input cardinality is small. But if the cardinality grows, the performance decreases. In order to provide an efficient way to join inputs with big cardinality, Tamino features the index-only join processing. This approach tries to join index entries instead of joining XML documents.

The following query shows a typical use case of the index-only join processing:

```
for   $b in input()/vendor,
    $a in input()/customer
    where $b/@vno = $a/vno
return
    <customers_of_vendor>
     { $b/vname  }
     { $b/@vno  }
     { $a/custname }
    </customers_of_vendor>
```

The query joins the two doctypes “vendor” and “customer”. The documents of the “vendor” doctype are not filtered by any predicate that can be exploited for creating an index access predicate. Moreover, the join predicate is an equal comparison on elements and attributes that occurs exactly once per document. By just joining the index entries, the expensive reading of a big number of documents can be avoided. This is particular true if you do not retrieve all results, but the first n or last n results by applying a position range filter:

```
(
for    $b in input()/vendor,
       $a in input()/customer
       where $b/@vno = $a/vno
     return
    <customers_of_vendor>
     { $b/vname  }
     { $b/@vno  }
     { $a/custname }
    </customers_of_vendor>
)
[position() > last() - 10]
```

The following list specifies the restrictions for join operations that can be processed index-only:

- Join is applied on expressions that retrieve a single node per document.
- Join predicates have to be specified in the where clause.
- Join predicates must be equal comparisons.

If a query does not satisfy the restrictions, join processing falls back to index-based nested-loop approach. Albeit the index-only join processing provides a tremendous performance improvement for a lot of queries, it is not always better than doing index-based nested-loop join. For a certain query it cannot be decided which approach is the better one by the Tamino XQuery processor. Therefore, the user can specify whether or not index-only processing is the appropriate join method. The user-switch to activate the index-only join processing is provided by the “optimization” of the Tamino XQuery pragma via the “join” parameter. The parameter can be set to “index-only” or “default” and activates or de-activates index-only join processing. This means, to activate index-only joining for our example query, it has to be stated like this:

```
{?optimization join="index-only"?}
(
for  $b in input()/vendor,
     $a in input()/customer
     where $b/@vno = $a/vno
return
    <customers_of_vendor>
     { $b/vname  }
     { $b/@vno  }
     { $a/custname }
    </customers_of_vendor>
)
[position() > last() - 10]
```

Index-based Processing of Aggregation Function

Tamino provides index-based processing of aggregate functions like min(), max(), count() and distinct-values().

Min and Max

The minimum and the maximum value of an element or attribute can be determined from a defined standard index. For example assuming the following query that retrieves the latest published books from a bibliography:

```
let $m := max(input()/bib/book/@year)
for $a in input()/bib/book
where $a/@year = $m
return $a
```

The latest publication year can be retrieved by accessing a standard index defined on the year attribute. The standard index on the year attribute is also used to retrieve all books that were published in the latest publication year.

Please note that the retrieval of the minimum and the maximum is not supported by a text index.

Count

In certain cases, a `fn:count()` expression can be optimized accessing an index or by just counting the documents in a doctype or collection. Assuming that in a bibliography each “bib” element contains exactly one “book” element, the following expression can be evaluated by just counting the number of documents in the “bib” doctype:

```
count(input()/bib/book)
```

If there is a standard index on the “title” element, the following query can be evaluated by just counting the number of matching documents found by the search predicate:

```
count(for $b in input()/bib/book where $b/title = "Data on the Web" return $b)
```

Distinct-values

A call of the distinct-values() function can be optimized by accessing an existing standard index. For example, assuming that there is a standard index defined on the “title” element, the following query can be evaluated by reading the values from the index:

```
distinct-values(input()/bib/book/title)
```

The index-based optimization is not possible for elements and attributes of type xs:string that have a collation defined.

Function Inlining

In order to minimize the overhead of user-defined functions, inlining is needed. This means that a function call is replaced by the code of the called function. The advantages are:

- Saving the overhead of function calls,
- Enabling optimizations across function boundaries.

Function inlining is not possible for recursive function calls. But due to the fact that inlining blows up the code of the calling query or function, it also might be problematic for non recursive functions. Since it is hard to find an optimal inlining strategy, the user is able to control the inlining behavior via the “inline” parameter of the “optimization” XQuery pragma. The argument of the “inline” parameter specifies the inlining strategy. The following strategies are available:

- None
- Default
- Full

The “default” strategy is in charge if no inlining is specified. The following query shows an example of how to apply “full” inlining to a query:

```
{?optimization inline="full"?}
import module namespace math = "http://example.org/math-functions";
math:power(2,2)
```

The None Inlining Strategy

If the value of the inlining parameter is “none”, no inlining is performed at all.

The Default Inlining Strategy

If the value of the inlining parameter is “default”, only those functions are inlined that have an “inline” hint. An “inline” hint is an XQuery processing instruction that is in front of a function declaration. The following module declaration provides an example of how to apply the “inline” hint.

```
module namespace math = "http://example.org/math-functions";
{?inline?}
declare function math:power($b as xs:integer, $e as xs:integer) as xs:integer
{
            if($e <= 0) then 1
            else math:power($b,$e - 1) * $b
}
```

If inlining is not possible, the “inline” hint is ignored.

The Full Inlining Strategy

When adhering to the full inlining strategy, all user-defined functions are inlined except for those which directly or indirectly reference themselves.

6 Efficient Queries: X-Query

The following sections present guidelines for efficient querying with X-Query:

Efficient X-Queries

X-Query processing involves a pre-selection and a post-selection step. In the pre-selection step, the indexes are used to select a subset of the final result set. In the post-processing step, this set is further restricted by applying the filter predicates that cannot be evaluated by an index access. This post-processing step involves the detailed analysis of each record contained in the intermediate result set.

If the preselection state is missing, it means that the whole doctype has to be read. Even for queries that have a small result this will cause a large response time. You can easily determine if a pre-selection is used if you include your query string into `ino:explain` (see next section **The X-Query Function** `ino:explain`). Tamino tells you whether your query involved pre-selection or/and post-processing.

Here are a few more guidelines for efficient querying:

- There is one situation when an indexed node cannot be handled during pre-selection: The query for the non-existence of the node. When a node does not exist, its value is also not contained in the index, and consequently, the test for a value cannot rely on the index. This test will therefore be processed during the post-processing phase. Depending on the size of the pre-selected document set, this test can be slow.
- A common problem is the use of the equality operator (=) when only a text index is defined, or the use of the contains operator (~=) when only a standard index is defined. In both cases, Tamino will correctly evaluate the query, but via post-processing. If you frequently apply both operators on the same node, consider defining it as both a standard *and* text index.
- Make use of Tamino's X-Query extensions to XPath. These expressions perform better than the equivalent standard XPath expressions.
- To always obtain correct results, make key and search expression type compatible, i.e. use a string search value for an alphanumeric key and a numeric search value for a numeric key. Comparing, for example, an alphanumeric constant with a numeric element causes the numeric element being converted into a string and a string comparison being performed. This would not return the expected results, and the performance suffers from this conversion, too.
- Generally, it should be considered whether queries using the contains operator (~=) should be reformulated, using the starts-with() operator. Here is an example:

```
/MyDocument [key-field ~= "abc*"]
```

...can be re-formulated as:

```
/MyDocument [starts-with(key-field, "abc"]
```

Starts-with makes use of a standard index, while the contains operator uses a text index. Usually, standard indexes consume less space and can therefore be loaded and updated faster. If the query needs post-processing, and if the key-field exists in many places in the document, the evaluation of the filter [key-field ~= "abc*"] may become costly.

A disadvantage, on the other hand, is the fact that many semantic differences exist: contains is more powerful, while starts-with requires the value of the key-field to start *exactly* with the given prefix: upper-/lower case must be observed, as well as the number of whitespace characters in the prefix value; there is no umlaut transformation in starts-with; and finally the contains operator looks for matching words within the value, while starts-with only checks the very beginning of the value.

- It is generally recommended to avoid wildcards (*) and descendant operators (//) if the path is known.

For further examples and more information, see the Advanced Concepts Guide on efficient querying.

Very Fast Queries

The following hints apply only for queries which need to be executed as fast as possible, meaning that it is important if they run 100 or 200 milliseconds (e.g. if they should run in parallel against a Tamino Server).

If you run queries via HTTP, it is possible with certain environments that the HTTP `GET` method has a better performance than the HTTP `POST` method. When constructing a high performance application, it may be useful to check the runtimes of a query with `GET` and `POST`.

Here is a list of hints for X-Query syntax:

- Queries must be indexed.
- Make sure that the result set is as small as possible.
- A list of logical terms in query expressions should be small. For example, a query with 20 `and/or` is slower than a query with one `and`.
- The logical terms with the highest hit rate (highest selectivity) should be placed at the beginning of the expression.
- Use the `betw` operator instead of `and` operations.

- Functions like `count()` in a query may be expensive.
- Avoid using the `adj` operator (see `tf:ContainsNearText`).

7 Query Processing Analysis

X-Query

The X-Query function `ino:explain` retrieves information about the execution plan of a query. The execution time of a query depends on the number and kind of processes that are needed to resolve a query. A query is processed in Tamino in the following order:

1. Query Parser
2. Query Optimizer
3. Processor-specific Optimizer
4. Index Processor
5. Postprocessor

A call to `ino:explain` provides information about which processing components are involved, to what degree the query can be optimized, and the work load of the index processor and postprocessor. With the information returned, you can rewrite your queries or update your schema to minimize processing costs. A detailed description of this X-Query function can be found in the X-Query Reference Guide under the topic *Functions*.

XQuery

If you add the `explain` directive to the query prolog of XQuery, the query will not be executed. Instead, an XML representation of the execution plan for the query expression is returned in the result document. You can use this information to determine which access paths will be used for Tamino data during query execution. Further details can be found in the XQuery 4 Reference Guide, chapter *Query Execution Plan*.

8 Suppressed Lookup of Index Entries

Tamino is delivered with a collection `ino:vocabulary`.

It contains the Doctype `ino:loadlist` which can contain any number of `ino:word` instances. The contents of an `ino:word` instance is any string (word) that may or may not be indexed. Typically, these are common “fill” words such as “and”, “it”, “is”, etc. or terms that occur frequently in the area in which Tamino is applied.

For entries in a load list, no check is made if index entries already exist from previous load operations, and the words are indexed anyway. This can speed up the indexing process when loading documents.

You can add multiple load lists and identify each one with the optional attribute `ino:loadlistname` on the `ino:loadlist` Node. When Tamino starts, the server loads and appends all the load lists, so that all load lists are always active when storing and indexing documents in Tamino.

9 Hardware Configuration

The overall Tamino performance is influenced by a number of components, such as CPU, virtual memory, physical memory (RAM), physical disk space, and network communications. Each of these may lead to performance bottlenecks. Increasing hardware performance with more CPU power and better I/O throughput should of course also improve response time. But before spending the money, you should find out where time consumption is highest: in the Tamino server, in the web server, on the network, or in the application.

In general, upgrading your hardware to improve performance is a step that you should only consider when you have achieved all of the possible performance gains using the methods described in the previous sections of this document.

Where to look?

On Windows, use the Performance Monitor, which is part of the Administrative Tools. Access it by choosing **Administrative Tools > Performance** from the Windows Control Panel. You can retrieve information on CPU usage, disk I/O, and memory usage.

Before you can use the Performance Monitor for I/O measurements, enable the `IO counting` by issuing the command `diskperf` at the command prompt. Note that this enables I/O monitoring after the next restart of your machine. `DISKPERF [-Y[E] | -N] [\\computername] -Y[E]` enables the disk performance counters when the system is restarted. `E` enables the disk performance counters used for measuring performance of the physical drives in a striped disk set when the system is restarted. Specify `-Y` without the `E` to restore the normal disk performance counters. `-N` disables the disk performance counters when the system is restarted. `\\computername` is the name of the computer you want to see or set disk performance counter use.

CPU

It is quite simple: If the CPU's workload is 100 %, you cannot get more speed out of your system. Additional CPUs can increase the throughput, unless the system is I/O bound. For small databases, almost everything can be kept in memory, according to the value of the buffer pool parameter of the database. In this case, an additional CPU is recommended if multiple users are working at the same time.

On a Windows platform, you should watch for the following on your Performance Monitor:

- When monitoring the CPU, check that the amount of CPU time spent in kernel mode is below 10 %. This is the counter **% Privileged time** of the **Processor** object. The instance to monitor is either **_Total** for a multiprocessor system, or **0** for a single CPU system.

If Tamino runs on multi-CPU machines (Symmetric Multi-Processing, SMP), automatic load balancing is carried out between CPUs. There is no load balancing between different machines, though.

Virtual Memory

Increasing the swapfile makes sense if Tamino fails with memory exception. Keep in mind that this may lead to swapping.

Disk I/O

Usually a disk controller can handle a specific amount of I/O operations or memory that can be transferred to or from the disk. If your system is I/O bound, it may help to add additional physical disks to the system in order to get a higher degree of parallelism. In order to do so, you have to distribute the Tamino data spaces over multiple disks. If you use multiple physical disks for your database and your system is I/O bound, it may be possible to increase the throughput by using multiple disk controllers. Some disk controllers support multiple channels, too. For recovery reasons, the data and index containers must be on a different physical disk than the log and backup data spaces. Additionally, it is recommended to put the index, the data container, and the journal space on separate disks. Using RAID (Redundant Array of Inexpensive [or Independent] Disks) can improve the overall I/O performance. A general recommendation in terms of high availability and performance is to have the log container on a RAID 1 (mirrored) disk and the data and index container on a RAID 5 disk.

On a Windows platform, the things to watch for on the Performance Monitor are:

- When the disk time is significant (about 50 %), you might want to increase the size of the buffer pool. If this does not help due to many update operations on the database, check which disks are the most active. Consider the following to distribute I/O load to different physical disks:
  1. Move database containers to locations stored on physically different disk drives.
  2. If you are using a server version, you might want to create disk volumes spanning multiple disks.

Tuning TCP/IP

If you have to serve a high load on your machine, you may run out of sockets due to the TCP_WAIT_TIMEOUT. This applies to the TCP_WAIT_TIMEOUT and the number of available sockets.

On Solaris, the command to show the time wait parameter is:

```
$ /usr/sbin/ndd /dev/tcp tcp_time_wait_interval
```

The command to set the parameter is:

```
$sudo /usr/sbin/ndd -set /dev/tcp tcp_time_wait_interval 30000
```

Tamino in a Multi-User Environment

The requirements for a multi-user environment are:

1. The server hardware should be powerful as possible: large RAM, many fast CPUs, powerful I/O system, etc. The more users you have, the more CPU power is necessary. Try to have fewer but faster CPUs to reduce locking conflicts.
2. Each single command (query or insert) should run as fast as possible. For information, see the section**Very Fast Queries**. Also, make use of cursors to keep the result set that is transferred from the server to an application small. However, keep locks and cursor only as long as you need them, so that they do not constrain the requests other users have.
3. The application process should not produce data of its own, e.g. protocol files, or at least as little data as possible. If data is produced, each application process should write them into separate data areas, for example directories or disks.

10 Performance Tuning - A Case Study

A case study conducted with a Tamino installation produced the following hitlist of factors for performance tuning:

1. Optimizing indexes for the most common queries
2. Optimizing query formulation
3. Optimizing efficiency of Java code calling Tamino APIs
4. Tuning operating system performance
5. Updating components of the system written by third parties or open source projects

The case exhibits several general principles for creating high-performance applications that have been observed repeatedly by Tamino users in the field:

- There is only one over-riding “design time” rule: Many small documents are more efficient than a small number of larger documents. This is due to fundamental design decisions by the developers of Tamino, based on analyses of how XML is used in the real world. It is sometimes necessary to write some code that resides between a data-producing application and Tamino that will decompose huge documents into more manageable chunks for efficient storage. For example, imagine a book that consists of dozens of chapters: storing each chapter as a separate document is more efficient both for Tamino itself and for most XML tools such as XSLT engines that you will use to work with the data after it is retrieved.
- Make it work, then make it fast. Trust Tamino to be fast, once properly tuned. If the application uses XML in a way that fits Tamino's design philosophy, do not worry about performance too much during the prototype phase.
- Tamino (and the same applies for almost all DBMS systems) is fastest when most of the work of satisfying the query request can be done by processing the indexes. Thus, the key to performance tuning is to ensure that indexes have been defined for the most frequently queried elements and attributes. Also, you can and should place indexes on the elements/attributes that reference join criteria. See *Efficient Querying* section in the *Advanced Concepts - From Schema to Tamino*

chapter of the Tamino documentation for more information, especially on using the “explain” facility.

- Remember that much of time a program spends retrieving data from Tamino into an application data structure may not be in the DBMS itself, but in the API. Be careful to use appropriate libraries and actual calls, depending on whether human time or machine time is the more precious resource in a given situation, since programmer convenience often comes at a performance price and vice-versa. For example, most programmers who are not XML experts will find DOM/JDOM APIs easier to use than lower-level event-driven interfaces such as SAX. The overhead of building a DOM tree, allocating memory to hold the values of the XML elements and attributes, and copying the data from Tamino to the API and then to the application, can be significant in some cases. Consider re-writing performance critical sections of an application in a way that uses the most efficient techniques to build application objects from the XML text retrieved from Tamino.
- Use system profiling tools to make sure that processing, disk, and memory resources are being used effectively by the overall system, of which Tamino is usually only one component. For example, a multiprocessor system will not be faster than a single processor system unless the various parts of the system can work in parallel, and this may require some profiling and load balancing. Similarly, make sure that the software is configured to use all the available memory if it is abundant, and to share it efficiently if it is not. Eliminate bottlenecks with hardware upgrades - additional memory, faster disk drives, faster networking - once they have been identified and if the hardware is cheaper than the human time or business cost of extensive tuning.
- Update to the latest version of available software. XML technologies are rapidly maturing and as XML is being used to power larger and larger, more and more performance-critical applications, developers are learning how to make it work faster and more reliably all the time. Using the latest version of software components allows one to benefit from the testing and tuning experience of other users.

Index