

# Exploit XML indexes for XML query performance in DB2 9

Skill Level: Intermediate

[Matthias Nicola \(mnicola@us.ibm.com\)](mailto:mnicola@us.ibm.com)  
DB2/XML Performance  
IBM

02 Nov 2006

Updated 29 Jul 2009

DB2® 9 provides pureXML storage and offers XQuery and SQL/XML as query languages. XML indexes are essential for high query performance, but their usage for query evaluation depends on how query predicates are formulated. This article presents a set of guidelines for writing XML queries and creating XML indexes in a consistent manner so that indexes speed up your queries as expected. Also learn what to look for in XML query execution plans to detect performance issues, and find out how to fix them. A downloadable "cheat sheet" summarizes the most important guidelines. *[2009 Jul 30: This article has been updated for DB2 9.5 and 9.7, including additional SQL/XML sample queries.--Ed.]*

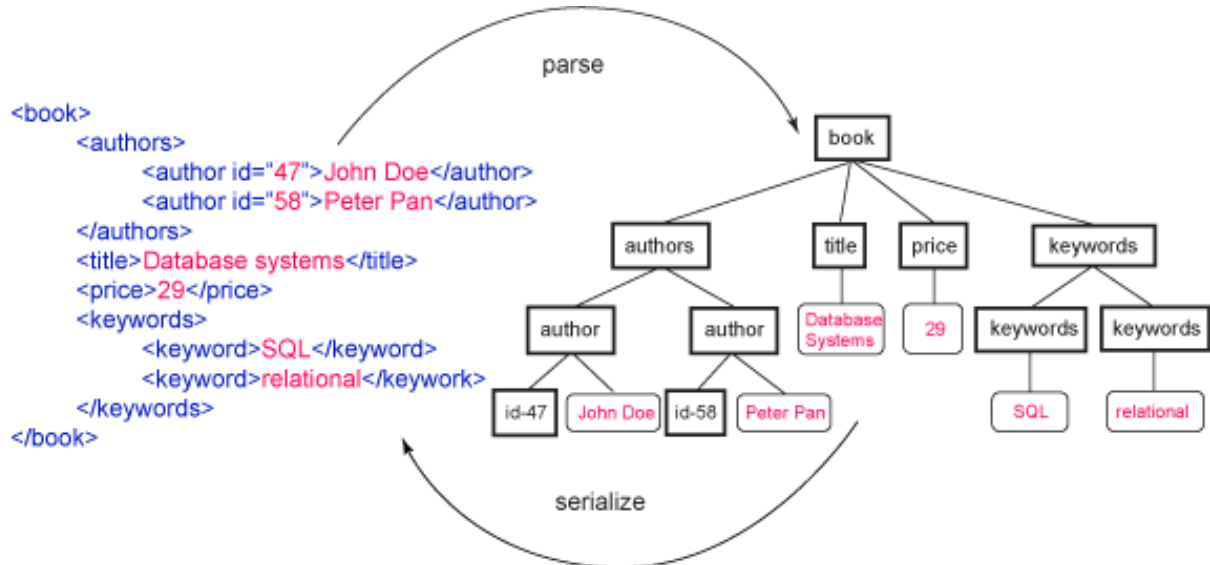
## Introduction

DB2 9 provides pureXML storage, along with XML indexes, XQuery and SQL/XML as query languages, XML schema support, and XML extensions to utilities such as Import/Export and Runstats. Just as for relational queries, indexes are crucial for high performance of your XQuery and SQL/XML statements. DB2 enables you to define *path-specific* XML indexes on XML columns. That means you can use them to index selected elements and attributes that are frequently used in predicates and joins. For example, using the sample data in [Figure 1](#), the following index `idx1` would be useful for look-ups and joins based on author IDs across all documents in the XML column `bookinfo` of the table `books`.

```
create table books(bookinfo XML);

create index idx1 on books(bookinfo)
generate keys using xmlpattern '/book/authors/author/@id'
as sql double;
```

**Figure 1. Sample XML document in textual (serialized) format, and parsed (hierarchical) format**



Since DB2 does not force you to associate a single XML schema with all documents in an XML column, the data types for specific elements and attributes are not known *a-priori*. Thus, each XML index requires you to specify a target type. You will see later in this article why the type matters. The available data types for XML indexes are:

- **VARCHAR(*n*)**: For nodes with string values of a known maximum length *n*.
- **VARCHAR HASHED**: For nodes with string values of arbitrary length. This index contains hash values of the actual strings and can be used for equality predicates only, not for range predicates.
- **DOUBLE**: For nodes with any numeric type.
- **DATE and TIMESTAMP**: For nodes with date or timestamp values.

The length of a VARCHAR(*n*) index is a hard constraint. If you insert a document where the value of an indexed element or attribute exceeds the maximum length *n*, the insert fails. Similarly, the `create index` statement for a VARCHAR(*n*) index fails if a value larger than *n* is encountered.

By default, the data types for DOUBLE, DATE, or TIMESTAMP indexes are not a

hard constraint. For example, the index `idx1` on author ID attributes is defined as `DOUBLE` since it is expected that these IDs are numeric values. If you insert a document where an author ID has the value "MN127", which is non-numeric, the document is still inserted, but the value "MN127" is not added to the index. This is correct and safe because the `DOUBLE` index can only evaluate numeric predicates that would never match the value "MN127" anyway. Thus, this value can safely be omitted from the index.

Since DB2 9.5 you can add the optional clause `REJECT INVALID VALUES` to your XML index definition. This clause enforces the `DOUBLE`, `DATE`, or `TIMESTAMP` type of the index as a hard constraint. If you define the following index, a document where an author ID has the value "MN127" cannot be inserted, and must not exist in the XML column when you create this index.

```
create index idx1 on books(bookinfo)
generate keys using xmlpattern '/book/authors/author/@id'
as sql double REJECT INVALID VALUES;
```

You can find more details on defining XML indexes in the "[DB2 pureXML Cookbook](#)". In the following discussion of XML index usage it is also assumed that you are familiar with the basic concepts of querying XML data in DB2. For more information, refer to the previous articles, "[Query DB2 XML Data with SQL](#)" (developerWorks, March 2006) and "[Query DB2 XML data with XQuery](#)" (developerWorks, April 2006) for an introduction, as well as "[pureXML in DB2 9: Which way to query your XML data?](#)" (developerWorks, June 2006) for more examples and details.

## XML index eligibility for XQuery and SQL/XML statements

Just as for relational queries, indexes are crucial for the high performance of your XQuery and SQL/XML statements. When your application submits a relational or XML query to DB2, the query compiler compares the query predicates with existing index definitions and determines if any available indexes can be used to execute the query. This process is known as "index matching" and produces a (possibly empty) set of eligible indexes for the given query. This set is input to the cost-based optimizer that decides whether to use any of the eligible indexes or not. In this article, the focus is on index matching rather than the optimizer's index selection. There is not much you can do about the optimizer decisions, except running "runstats" to provide the optimizer with accurate statistics about your data. However, there is a lot you can do to ensure index matching.

Index matching is usually trivial in the relational case. DB2 can use an index defined on a single relational column to answer any equality or range predicate on this column. But, for XML columns this is more complex. While an index on a relational column contains all values from that column, an XML index contains only values of

nodes that match both the XML pattern *and* the data type in the index definition. Thus, an XML index can be used to evaluate an XML query predicate only if this index is of the "right" data type *and* contains at least all XML nodes that satisfy the predicate. Thus, there are two key requirements for XML index eligibility:

1. The XML index definition is equally or less restrictive than the query predicate ("containment").
2. The data type of the index matches the data type in the query predicate.

This article explains how to design your XML indexes and queries to ensure that these requirements are met, and how to avoid common pitfalls. This starts with understanding your queries' execution plans. The existing explain tools in DB2, such as Visual Explain and db2exfmt, can be used to view query execution plans for XQuery and SQL/XML just as they can for traditional SQL.

## XML query evaluation: Execution plans and new operators

To execute XML queries, DB2 9 introduces three new internal query operators called XSCAN, XISCAN, and XANDOR. Together with existing query operators (such as TBSCAN, FETCH, and SORT) these new operators allow DB2 to generate execution plans for SQL/XML and XQueries. Now take a look at the three new operators and how they work together with XML indexes in a query execution plan.

### **XSCAN (XML Document Scan)**

DB2 uses the XSCAN operator to traverse XML document trees and, if needed, to evaluate predicates and extract document fragments and values. XSCAN is *not* an "XML table scan" but it can appear in an execution plan *after* a table scan to process each of the documents.

### **XISCAN (XML Index Scan)**

Like the existing relational index scan operator for relational indexes (IXSCAN), the XISCAN operator performs lookups or scans on XML indexes. The XISCAN takes a value predicate as input, such as a path-value pair like `/book[price = 29]` or `where $i/book/price = 29`. It returns a set of row IDs and node IDs. The row IDs identify the rows that contain the qualifying documents, and the node IDs identify the qualifying nodes within these documents.

### **XANDOR (XML Index AND'ing)**

The XANDOR operator evaluates two or more equality predicates simultaneously by driving multiple XISCANs. It returns the row IDs of those documents that satisfy all of these predicates.

Now look at a sample query (in equivalent XQuery and SQL/XML notation) to

understand its execution plan without an index, with one index, and with multiple indexes:

```
-- XQuery:
xquery for $i in db2-fn:xmlcolumn("BOOKS.BOOKINFO")
where $i/book/title = "Database systems" and $i/book/price = 29
return $i/book/authors;

-- SQL/XML:
select XMLQUERY('$i/book/authors' passing bookinfo as "i")
from books
where XMLEXISTS('$i/book[title = "Database systems" and price =
29]'
                passing bookinfo as "i");

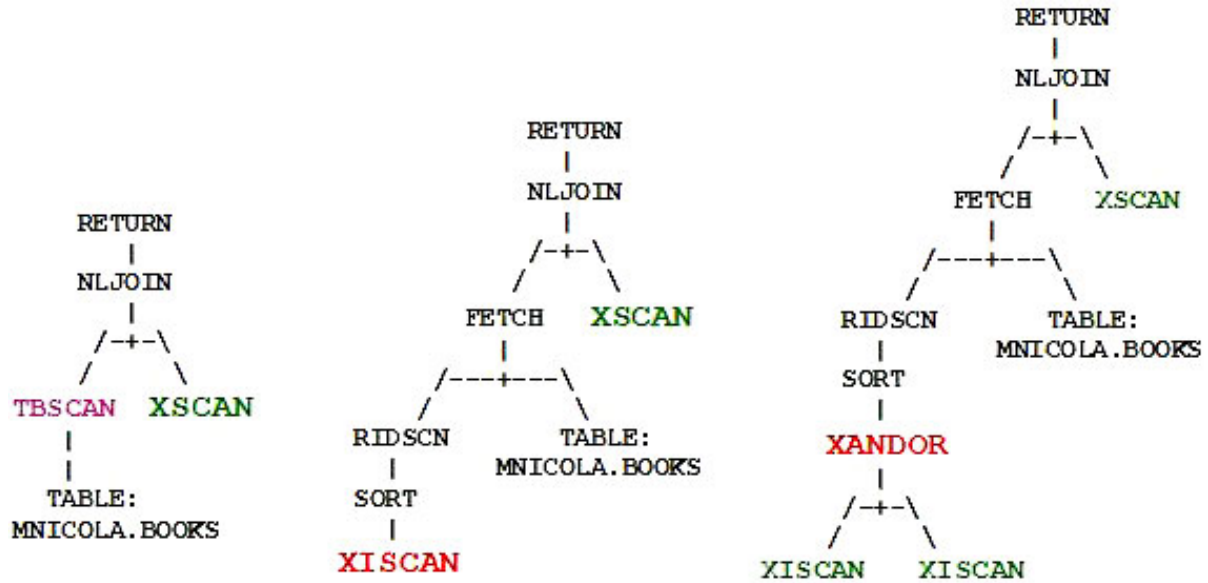
-- Two Indexes:
create index idx1 on books(bookinfo) generate keys
using xmlpattern '/book/title' as sql varchar(50);

create index idx2 on books(bookinfo) generate keys
using xmlpattern '/book/price' as sql double;
```

In Figure 2, you see three different execution plans for this query (simplified output from db2exfmt). It's best to read such execution plans from the left-lowest operator in the tree, since the logical flow in the plan is from bottom to top and left to right.

The leftmost plan (a) is used if there is no eligible index for the predicates in this query. The table scan operator (TBSCAN) reads *all* rows from the table "BOOKS." For each row, the nested loop join (NLJOIN) operator passes a pointer to the corresponding XML document to the XSCAN operator. As such, the NLJOIN does not act as a classical join with two input legs, but facilitates access to the XML data for the XSCAN operator. The XSCAN operator traverses each document, evaluates the predicates, and extracts the "authors" element if the predicates are satisfied. The RETURN operator completes the query execution and returns the query result to the API.

**Figure 2. Three execution plans, (a) no index, (b) one index, (c) two indexes**



(a) No Index

(b) One Index

(c) Two Indexes

If you have an index for one of the two predicates, for example index `idx1` on `/book/price`, you will see an execution plan similar to plan (b) in Figure 2. The XISCAN probes the index with the path-value pair `(/book/price, 29)` and returns the row IDs for documents where the price is 29. These row IDs are sorted to remove duplicates (if any) and optimize the subsequent I/Os to the table. The row ID scan (RIDSCN) operator then scans these row IDs, triggers row prefetching, and passes the row IDs to the FETCH operator. For each row ID, the FETCH operator reads the corresponding row from the table. The benefit of this plan is that only a fraction of the rows in the table are retrieved, meaning only those where "price" is 29. This is a lot cheaper than a full table scan that reads every row. For each row fetched, the XSCAN operator processes the corresponding XML document. It evaluates the predicate on "title" and, if the predicate is satisfied, extracts the "authors" element. There can be many documents where this second predicate is not true, and the XSCAN may still perform a lot of work to weed them out. Thus, you may see even better performance if the second predicate was covered by an index too.

If you have indexes for both predicates you may see execution plan (c) in Figure 2. This plan uses two XISCANs, one for each predicate and index. The XANDOR operator uses these XISCANs to alternately probe into the two indexes to efficiently find the row IDs of the documents that match *both* predicates. The FETCH operator then only retrieves these rows, thus minimizing I/O to the table. For each document, the XSCAN subsequently extracts the "authors" element. If your predicates include `//` or `*` in the path, or if you use range comparisons (such as `<` and `>`), you will see the index AND'ing (IXAND) operator instead of the XANDOR. Logically, both perform the same job but for different types of predicates and with different optimizations.

The optimizer can decide to not use an index even if it could be used. For example, the optimizer may choose plan (b) over plan (c) if the second index does not significantly reduce the number of rows retrieved from the table, such as if the cost of accessing the index outweighs the savings in I/O to the table. Yet, you want to make sure that the optimizer considers all eligible indexes to then pick the plan with the lowest cost and shortest execution time. In other words, you want to observe the two requirements for XML index eligibility:

- The XML index contains at least all XML nodes that satisfy the predicate.
- The data type in the query predicate and the index definition are compatible.

## Wildcards in XML indexes and query predicates

The wildcards `//` and `*` can affect the containment relationship between an index and a query predicate. This is because path expressions such as `/book/price` and `//price` are different. The path `/book/price` identifies all `price` elements that are immediate children of the element "book." But, the path `//price` identifies `price` elements anywhere at any level of an XML document. Thus, `/book/price` identifies a subset of the elements specified by `//price`. It is said that `//price` "contains" `/book/price` but not the other way round.

Now look how that affects index eligibility. Take the following query as an example. It shows four variations of its where clause in Table 1.

```
XQUERY
for $i in db2-fn:xmlcolumn("BOOKS.BOOKINFO")
where $i/book/price = 29
return $i/book/authors
```

The two rightmost columns in Table 1 represent two alternative index definitions, and the rows in the table show which of the predicates can (+) or cannot (-) be evaluated by either of the indexes. Now step through the rows in Table 1 to explore index eligibility for each predicate.

For the first predicate, the index on `/book/price` is ineligible because it only contains "price" elements that are immediate children of "book." The index does not contain "price" elements at a deeper level, which may exist in the table and are potential matches for the predicate path `$i//price`. Thus, if DB2 used the index on `/book/price`, it might return an incomplete result. The second index, on `//price`, is eligible since it contains all `price` elements at any level of the document, as required for the predicate.

The third predicate uses the star (`*`) as a wildcard such that it looks for *any* child

element under "book" with a value of 29. Not only "price" elements can fulfill this predicate. For example, a document with value 29 in the element `/book/title` would be a valid match. But, `title` elements are not included in either of the two indexes in Table 1. Therefore, neither index can be used because DB2 might return incomplete results for this predicate.

**Table 1. Index eligibility with wildcards in XML indexes and predicates**

#	Predicate/Index Definition	...using xmlpattern '/book/price' as sql double;	...using xmlpattern '//price' as sql double;
1	where <code>\$i//price = 29</code>	-	+
2	where <code>\$i/book/price = 29</code>	+	+
3	where <code>\$i/book/* = 29</code>	-	-
4	where <code>\$i/*/price = 29</code>	-	+

The fourth predicate `$i/*/price = 29` looks for price elements under *any* root element, not just "book." If there is a document with a path `/journal/price`, then it might satisfy the predicate `$i/*/price = 29`, but it would not be included in the index on `/book/price`. Therefore, this index cannot be used because DB2 would again risk returning an incomplete query result. However, the index on `//price` contains any price element, irrespective of the root element.

In a nutshell, the DB2 query compiler always needs to be able to prove that the index is equally or less restrictive than the predicate, so that it contains everything that the predicate is looking for.

Be aware that using wildcards in index definitions *may* inadvertently index more nodes than needed. Wherever possible, it is recommended to use the exact path to the desired elements or attributes in index definitions and queries, without wildcards. Very generic XML index patterns such as `//*` or `//text()` are possible, but should be used with caution. An index on `//*` would even index non-leaf elements, which is typically not useful and can easily exceed the length constraint of a `Varchar(n)` index.

## Namespaces in XML indexes and query predicates

XML index eligibility requires your attention if namespaces are involved. First of all, if the XML documents in your table contain namespaces, then the index definition will need to take that into account. This again comes down to index/predicate containment. Take the following XML document and index definition as an example:

```
<bk:book xmlns:bk="http://mybooks.org">
```

```
<bk:title>Database Systems</bk:title>
<bk:price>29</bk:price>
</bk:book>
```

```
CREATE INDEX idx3 ON books(bookinfo)
GENERATE KEYS USING XMLPATTERN '/book/price' AS SQL DOUBLE;
```

This index `idx3` does not contain any index entries for this sample document because it is defined to be an index for `/book/price` elements with an empty namespace. But, any of the following index definitions could be used to index the price element properly:

```
CREATE INDEX idx4 ON books(bookinfo) GENERATE KEYS USING
XMLPATTERN
'declare namespace bk="http://mybooks.org"; /bk:book/bk:price'
AS SQL DOUBLE

CREATE INDEX idx5 ON books(bookinfo) GENERATE KEYS USING
XMLPATTERN
'declare default element namespace "http://mybooks.org";
/book/price' AS SQL DOUBLE

CREATE INDEX idx6 ON books(bookinfo) GENERATE KEYS USING
XMLPATTERN
'/*:book/*:price' AS SQL DOUBLE
```

Index `idx4` declares the namespace and prefix explicitly to match the document. Index `idx5` declares the namespace as a default namespace, and therefore does not use a prefix in the XML pattern `/book/price` since the default namespace is implied. Index `idx6` simply uses wildcards to match any namespace. The same options are available when you formulate predicates in XQuery or SQL/XML statements:

#### Query 4:

```
-- XQuery:
XQUERY declare namespace bk="http://mybooks.org";
for $b in db2-fn:xmlcolumn("BOOKS.BOOKINFO")/bk:book
where $b/bk:price < 10
return $b

-- SQL/XML:
select bookinfo
from books
where XMLEXISTS('declare namespace bk="http://mybooks.org";
$b/bk:book[bk:price < 10]'
passing bookinfo as "b")
```

#### Query 5:

```
-- XQuery:
XQUERY declare default element namespace "http://mybooks.org";
for $b in db2-fn:xmlcolumn("BOOKS.BOOKINFO")/book
```

```

where $b/price < 10
return $b

-- SQL/XML:
select bookinfo
from books
where XMLEXISTS('declare default element namespace
"http://mybooks.org";
$b/book[price < 10]'
passing bookinfo as "b")
    
```

**Query 6:**

```

-- XQuery:
XQUERY for $b in db2-fn:xmlcolumn("BOOKS.BOOKINFO")/*:book
where $b/*:price < 10
return $b

-- SQL/XML:
select bookinfo
from books
where XMLEXISTS('$b/*:book[*:price < 10]'
passing bookinfo as "b")
    
```

Table 2 has one row for each of these three queries, and one column for each of the indexes `idx3` through `idx6` defined above. You can make several important observations in Table 2. First, `idx3` without a namespace cannot be used for any queries that consider namespaces. Next, you see that the rows for query 4 and query 5 have identical entries, and the columns for indexes `idx4` and `idx5` also have identical entries. This is because the explicit namespace definitions and the default namespace definitions are logically the same, and just a different notation for the same thing. You can use either one without affecting index matching. Index `idx6` with namespace wildcards is eligible for all of your sample queries, and it could even be used for a predicate without a namespace, such as `$b/price < 10`. Index `idx6` is also the only index that matches the predicate in query 6. Indexes `idx4` and `idx5` contain index entries for one specific namespace but cannot be used for query 6 that looks for book prices in *any* namespace. Thus, the containment requirement is violated.

**Table 2. Index eligibility with namespaces in XML indexes and predicates**

#	Query/Index Definition	idx3 (no namespace)	idx4 (explicit namespace)	idx5 (default namespace)	idx6 (namespace wildcard)
1	Query4 (explicit namespace)	-	+	+	+
2	Query 5 (default namespace)	-	+	+	+
3	Query 6	-	-	-	+

(namespace wildcard)

## Data types in XML indexes and query predicates

In addition to proper index or predicate containment with wildcards and namespaces, the second requirement for index eligibility is that data types of predicates and indexes need to match. In all the previous examples, the `/book/price` element is always indexed as DOUBLE. But, you could decide to index book prices as VARCHAR, as shown in Table 3. However, note that value predicates also have a data type that is determined by the type of the literal value. A value in double quotes is always a string, but a numeric value without quotes is interpreted as a number. As you see in Table 3, a string predicate can only be evaluated with an XML index of type VARCHAR, while a numeric predicate can only be evaluated with an index of type DOUBLE.

The data type of relational indexes is always determined by the type of the indexed column. However, since DB2 does not force you to associate an XML schema with an XML column, the data types for elements or attributes are not predetermined. Thus, each XML index requires a target type. And the type matters. Assume a price element has the value 9. A string predicate `"9" < "29"` is false, while a numeric comparison `9 < 29` is true. This underlines that you should use DOUBLE indexes if you want semantically correct numeric comparisons. The "price" element is likely best indexed as DOUBLE.

**Table 3. Data types in XML indexes and predicates**

#	Predicate or index definition	...using xmlpattern '/book/price' as sql double;	...using xmlpattern '/book/price' as sql varchar(10);
1	where \$i/book/price < "29"	-	+
2	where \$i/book/price < 29	+	-

## Use indexes for XML join predicates

In the previous examples, you saw value predicates that include literal values. These literal values determine the data type of the comparison. Such a determination is usually not possible for join predicates. Assume you have a table "authors" with detailed author information in an XML format that includes the author IDs that also occur in your book data. You want to use a join to retrieve the detailed author data only for authors of books in the books table. Defining indexes on the author IDs seems useful:

```

create table books (bookinfo xml);
create table authors (authorinfo xml);

create index authorIdx1 on books(bookinfo) generate key using
xmlpattern '/book/authors/author/@id' as sql double;

create index authorIdx2 on authors(authorinfo) generate key
using
xmlpattern '/author/@id' as sql double;

XQUERY
for $i in db2-fn:xmlcolumn("BOOKS.BOOKINFO")
  for $j in db2-fn:xmlcolumn("AUTHORS.AUTHORINFO")
where $i/book/authors/author/@id = $j/author/@id
return $j;

```

This query retrieves the desired author information, but it would not use either of the indexes for the join processing. Note that the join predicate on the author ID does not contain a literal value that would indicate the data type of the comparison. Therefore, DB2 needs to consider matching author IDs of any data type. For example, consider the book and author data in Table 4. Author John Doe has a numeric ID value (47), while author Tom Noodle has a non-numeric ID value (TN28). Both have valid matches in the other table. Therefore, both need to be included in the result of the join. However, if DB2 used the numeric indexes **authorIdx1** or **authorIdx2**, it would not find author ID "TN28" and return an incomplete join result. Thus, DB2 cannot use those indexes and resorts to a table scan to ensure a correct query result.

**Table 4. Sample book and author data**

Book	Author
<pre> &lt;book&gt;   &lt;authors&gt;     &lt;author id="47"&gt;John Doe&lt;/author&gt;   &lt;/authors&gt;   &lt;title&gt;Database Systems&lt;/title&gt;   &lt;price&gt;29&lt;/price&gt; &lt;/book&gt; </pre>	<pre> &lt;author id="47"&gt;   &lt;name&gt;John Doe&lt;/name&gt;   &lt;addr&gt;     &lt;street&gt;555 Bailey Av&lt;/street&gt;     &lt;city&gt;San Jose&lt;/city&gt;     &lt;country&gt;USA&lt;/country&gt;   &lt;/addr&gt;   &lt;phone&gt;4084511234&lt;/phone&gt; &lt;/author&gt; </pre>
<pre> &lt;book&gt;   &lt;authors&gt;     &lt;author id="TN28"&gt;Tom Noodle&lt;/auth&gt;   &lt;/authors&gt;   &lt;title&gt;International Pasta&lt;/title&gt;   &lt;price&gt;19.95&lt;/price&gt; &lt;/book&gt; </pre>	<pre> &lt;author id="TN28"&gt;   &lt;name&gt;Tom Noodle&lt;/name&gt;   &lt;addr&gt;     &lt;street&gt;213 Rigatoni Road&lt;/street&gt;     &lt;city&gt;Toronto&lt;/city&gt;     &lt;country&gt;Canada&lt;/country&gt;   &lt;/addr&gt;   &lt;phone&gt;4162050745&lt;/phone&gt; &lt;/author&gt; </pre>

However, in many situation you probably don't have a mix of numeric and

non-numeric values in a given element or attribute of your documents. If you know that all author IDs are numbers, you can indicate this in your query to allow DB2 to use the DOUBLE indexes. The following query explicitly casts both sides of the join predicate to DOUBLE. This asks for a numeric comparison only and explicitly disregards non-numeric join matches. Therefore, DB2 can use a DOUBLE index for faster join processing.

```
XQUERY
for $i in db2-fn:xmlcolumn("BOOKS.BOOKINFO")
  for $j in db2-fn:xmlcolumn("AUTHORS.AUTHORINFO")
where $i/book/authors/author/@id/xs:double(.) =
$j/author/@id/xs:double(.)
return $j;
```

Since this query contains no value predicate to restrict either the books or the authors table, DB2 has to perform a table scan on either of the two tables to read all author IDs. For each author ID, an index is then used to probe whether that ID occurs in the other table. This is a lot faster than a nested loop join of two table scans without any index usage. DB2's cost-based optimizer decides which table to scan and which table to access through the index. Table 5 shows these two alternative execution plans. These two execution plans are also possible if you write the same join in SQL/XML notation in one of the following two ways:

#### Query 1:

```
select authorinfo
from books, authors
where xmlexists('$b/book/authors[author/@id/xs:double(.) =
$a/author/@id/xs:double(.)]'
passing bookinfo as "b", authorinfo as "a");
```

#### Query 2:

```
select authorinfo
from books, authors
where xmlexists('$a/author[@id/xs:double(.) =
$b/book/authors/author/@id/xs:double(.)]'
passing bookinfo as "b", authorinfo as "a");
```

Note that Query 1 and Query 2 differ in the "direction" of the XMLEXISTS predicate. In both queries, the join predicate is expressed within the square brackets. In Query 1, the predicate in square brackets is a predicate on the expression starting with \$b, so it is a predicate on the `books` table. In Query 2, the join condition is a predicate on the expression starting with \$a; that is, a predicate on the `authors` table. DB2 9.7 disregards this syntactical difference and chooses the cheaper of the two execution plans in Table 5, based on cost and cardinality estimates.

However, prior to DB2 9.7 the "direction" of the XMLEXISTS predicate in Query 1

and Query 2 determines which of the two execution plans in Table 5 is used. Since Query 1 expresses the join predicate on the `books` table, DB2 9.1 and 9.5 perform a table scan on `authors` and then use the index `AUTHORIDX1` to probe into the `books` table. This is shown on the left in Table 5.

Query 2 applies the join predicate to the `authors` table. Hence, DB2 9.1 and 9.5 perform a table scan on `books` and then use the index `AUTHORIDX2` to probe into the `authors` table (right side of Table 5). Thus, the way you write the `XML EXISTS` predicate can affect the join order in these previous versions of DB2. If table scans cannot be avoided, try to have on the smaller table.

**Table 5. Execution plans for XML join queries, produced by db2exfmt**

Query 1				Query 2			
<pre> Rows RETURN ( 1) Cost I/O   3.59881e-005 NLJOIN ( 2) 5410.62 743 /-----\ 1.29454e-007      278 NLJOIN          NLJOIN ( 3)            ( 6) 4311.96        1098.66 570            173 /-----\      /-+ \ 556            139      2 TBSCAN        XSCAN    FETCH    XSCAN ( 4)          ( 5)      ( 7)      ( 1) 106.211      7.5643    47.237   7.56 14           1         34       1                                   556         139       556 TABLE: MATTHIAS  RIDSCN  TABLE: MA AUTHORS        ( 8)   BOOK 15.2133 2   139 SORT ( 9) 15.2129 2   139 XISCAN ( 10) 15.1542 2   556 XMLIN: MATTHIAS AUTHORIDX1                     </pre>				<pre> Rows RETURN ( 1) Cost I/O   8.37914e-015 NLJOIN ( 2) 5410.63 743 /-----\ 1.29454e-007      6.47269e-008 NLJOIN          NLJOIN ( 3)            ( 6) 4311.96        1098.67 570            173 /-----\      /-+ \ 556            139      4.65661e-010 TBSCAN        XSCAN    FETCH    XSCAN ( 4)          ( 5)      ( 7)      ( 11) 106.211      7.56429   47.2365   7.5643 14           1         34       1                                   556         139       556 TABLE: MATTHIAS  RIDSCN  TABLE: MATTHIAS BOOKS           ( 8)   AUTHORS 15.2128 2   139 SORT ( 9) 15.2124 2   139 XISCAN ( 10) 15.1537 2   556 XMLIN: MATTHIAS AUTHORIDX2                     </pre>			

To summarize the advice for XML join queries, always cast join predicates to the type of the XML index that should be used. Otherwise, the query semantics do not allow index usage. If the XML index is defined as DOUBLE, you cast the join predicate with `xs:double`. If the XML index is defined as VARCHAR, you cast the join predicate with `fn:string`, and so forth as you see in Table 6. (Strictly speaking, DB2 9.7 does not necessarily require the use of `fn:string` anymore to enable VARCHAR indexes for join predicates. But, it's still required in 9.5 and it doesn't hurt to be specific in your predicates.)

**Table 6. Casting join predicate to allow XML index usage**

Index SQL Type	Cast Join Predicate Using:	Comment
DOUBLE	<code>xs:double</code>	For any numeric comparison
VARCHAR(n), VARCHAR HASHED	<code>fn:string</code>	For any string comparison
DATE	<code>xs:date</code>	For date comparison
TIMESTAMP	<code>xs:dateTime</code>	For timestamp predicates

## Index support for "between" predicates

XQuery does not have a special function or operator similar to the relational "between" predicate. Additionally, the existential nature of the XQuery general comparison predicates requires attention when you express a "between" condition.

Assume you want to find books with a price between 20 and 30. You might intuitively use the predicate `/book[price > 20 and price < 30]`, but it doesn't constitute a "between predicate." This means that if you have an index on `/book/price`, DB2 cannot perform an index range scan from 20 to 30 to find books in that price range. This is because a book document could possibly have multiple price children, as in the following example:

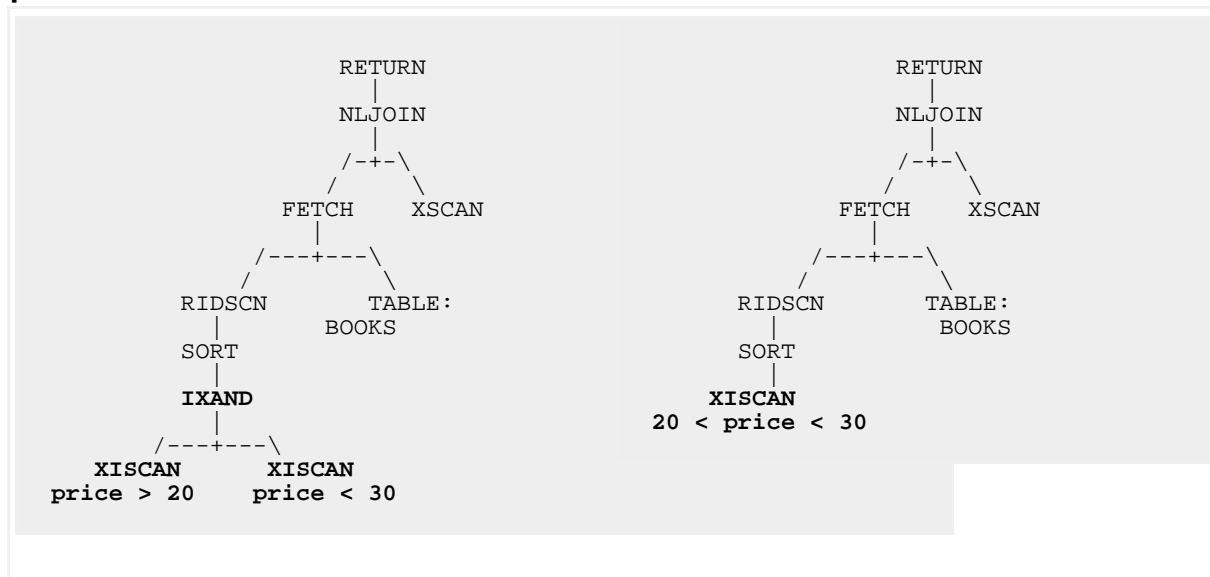
```
<book>
  <title>Database Systems</title>
  <price currency="RMB">40</price>
  <price currency="USD">10</price>
</book>
```

Since the general comparisons (`>`, `<`, `=`, `<=`, etc.) have existential semantics, the predicate `/book[price > 20 and price < 30]` selects a book element if there exists a child element "price" with a value larger than 20, and if there exists a child element "price" with value less than 30. These can be one and the same element or two different "price" elements. The sample document above satisfies the predicate because there is a price greater than 20 and there is also a (different) price less than 30. Yet, none of the two prices are between 20 and 30.

If DB2 used a single index range scan from 20 to 30, it would miss this document and return an incomplete query result. Instead DB2 needs to compute the intersection two index scans, which is often significantly more costly. The difference in execution plans is shown in Figure 3. The execution plan on the left shows the index AND'ing plan that DB2 would have to consider to catch your sample document. This plan is not efficient because both XISCANS produce a potentially very large number of row IDs out of which many need to be eliminated by the IXAND operator above. This is because many of the books are over \$20, and many are under \$30. In fact, both XISCANS combined produce more row IDs than rows in the table. This requires heavy work in the IXAND operator only to find a potentially small intersection.

If your intention is a real "between" predicate, then the execution plan on the right is much better, because a single range scan with a start-top predicate delivers the matching row IDs only. There is much less index access and no index AND'ing so that the overall performance can be one or two order of magnitude better - depending on predicate selectivity.

**Figure 3. Index AND'ing vs. single range scan to evaluate a pair of range predicates**



A pair of range predicates on the same data item can be interpreted by the DB2 compiler as a "between," and evaluated by a single index range scan, only if DB2 can deduce that the item is a singleton and not a sequence of more than one item. In other words, the predicate needs to be formulated such that both parts of the between (the > and the < ) are always applied to the same single item. This can be achieved by value comparisons (>, <, =, and so on), the self axis, or attributes.

**Value comparison**

If you know that a book has at most one price element, you can write the query

using XQuery value comparisons, which force the comparison operands to be singletons. For example, `/book[price gt 20 and price lt 30]` can safely be interpreted as a "between," and evaluated by a single range scan of the price index. If a book with more than one price child element is encountered, the query fails at runtime with an error.

### Self axis

As an alternative to value comparisons, you can use the self axis (denoted by a dot ".") to express a "between" predicate. The self axis in the expression `/book/price[. > 20 and . < 30]` ensures that that both predicates apply to the same price element. This constitutes a "between" predicate since the self axis always evaluates to a singleton. This predicate allows a book to have multiple prices but requires all of them to have a value between 20 and 30. The advantage over using value comparisons is that you don't risk a runtime error.

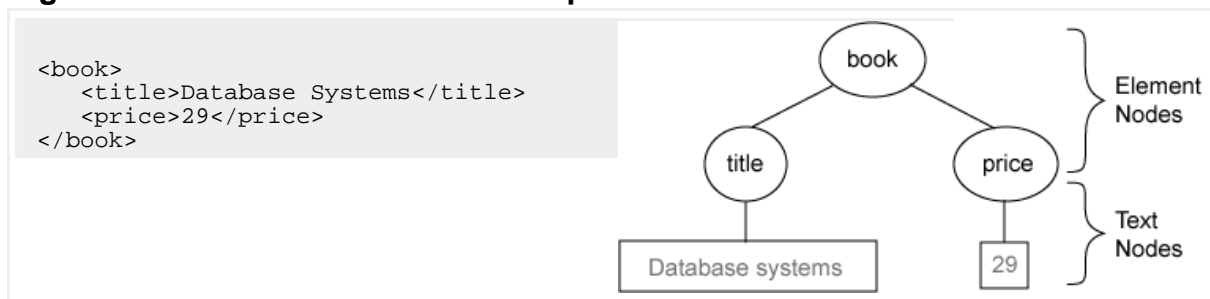
### Attributes

If the book price happens to be an attribute, then it can occur at most once per book element. In the expression `/book[@price>20 and @price<30]` the operands of the range predicates are singletons, so DB2 can perform a single index range scan to evaluate the "between."

## Indexing text nodes and the XPath step `/text()`

Briefly recall what *text nodes* are. Figure 4 shows a sample document and its hierarchical format in the XML data model. Each element is represented by an element node, and the actual data values are represented by text nodes. In the XML data model, the value of an element is defined as the concatenation of all text nodes in the subtree under that element. Thus, the value of the element "book" is "Database Systems29." The value of an element at the lowest level is equal to its text node, for example the value of the element "price" is "29."

**Figure 4. XML data model of the sample document**



The XPath expressions `/book/price` and `/book/price/text()` are different. The former identifies the element node "price", the latter points to the text node with the value "29". Thus, the following two queries return different results.

```
XQUERY for $b in db2-fn:xmlcolumn("BOOKS.BOOKINFO")/book
return $b/price

XQUERY for $b in db2-fn:xmlcolumn("BOOKS.BOOKINFO")/book
return $b/price/text()
```

The first query returns the full element node, i.e. `<price>29</price>`, while the second query only returns its text node value 29. If an XPath expression without `/text()` is used in a query predicate, such as `$b/book` in the following query, DB2 automatically uses the element's value to evaluate the predicate. Since the value of the "book" element is "Database Systems29", this query would return the sample document as a valid match.

```
XQUERY for $b in db2-fn:xmlcolumn("BOOKS.BOOKINFO")
where $b/book = "Database Systems29"
return $b
```

But, the next query, where you added `/text()` to the path in the where clause, does not return your sample document. This is because there is no text node immediately under the element "book."

```
XQUERY for $b in db2-fn:xmlcolumn("BOOKS.BOOKINFO")
where $b/book/text() = "Database Systems29"
return $b
```

Thus, in the general case, the query semantics are different depending on the use of `/text()`. Elements at the lowest level that only have a single text node *may* show the same behavior with and without `/text()`. For example, the next two queries may return the same result, but *only* if all "price" elements encountered during the query execution have a single text node and no other child nodes.

```
XQUERY for $b in db2-fn:xmlcolumn("BOOKS.BOOKINFO")
where $b/book/price < 10
return $b

XQUERY for $b in db2-fn:xmlcolumn("BOOKS.BOOKINFO")
where $b/book/price/text() < 10
return $b
```

Since `/text()` generally makes a difference for query semantics, it also makes a difference for index eligibility. In Table 7 you see that a predicate with `/text()` can only be evaluated by an index that also specifies `/text()` in its XML pattern. If your index does not use `/text()`, your predicates should also not use `/text()`.

**Table 7. Indexes and predicates with and without `/text()`**

Predicate or index definition	...using xmlpattern '/book/title/text()' as sql	...using xmlpattern '/book/title' as sql
-------------------------------	--	---

	<code>varchar(128);</code>	<code>varchar(128);</code>
where <code>\$/i/book/title = "Database Systems"</code>	-	+
where <code>\$/i/book/title/text() = "Database Systems"</code>	+	-

For simplicity it is recommend that you do not use `/text()` in either XML index definitions or query predicates. It may be tempting to define an index on the XML pattern `//text()` to support predicates for any path expressions ending in `/text()`. However, such an index contains *all* text node values from *all* documents in the XML column. Hence, the index will be very big and costly to maintain during insert, update, and delete operations. You should generally avoid such an index unless your application is mostly read-only and you really cannot predict which elements will be used in search conditions.

## Indexing non-leaf elements

In the previous section, you have seen a predicate on the element `/book` that is called a non-leaf (or non-atomic) element, since it contains other elements. Although you can define indexes on non-leaf elements, they are useful only in rare cases. Consider the following XML document. An index on the XML pattern `/book` would contain a single index entry for this document, and the value of that index entry is "John DoePeter PanDatabase Systems29SQLrelational." This is not useful since queries would typical not use such concatenated values in their predicates. Most indexes are always on leaf elements.

```
<book>
  <authors>
    <author id="47">John Doe</author>
    <author id="58">Peter Pan</author>
  </authors>
  <title>Database Systems</title>
  <price>29</price>
  <keywords>
    <keyword>SQL</keyword>
    <keyword>relational</keyword>
  </keywords>
</book>
```

There are a few cases where indexes on non-leaf elements can make sense. For example, assume your queries contain predicates on area codes and on full phone numbers. In that case, you could choose to design your phone elements as shown in this document.

```
<author id="47">
  <name>John Doe</name>
  <phone>
```

```
<areacode>408</areacode>
<number>4511234</number>
</phone>
</author>
```

Then, you can define one XML index on the non-leaf element "phone" and one on the element "areacode":

```
create index phoneidx on authors(authorinfo) generate key using
xmlpattern '/author/phone' as sql double;

create index areaidx on authors(authorinfo) generate key using
xmlpattern '/author/phone/areacode' as sql double;
```

This will allow both of the following queries to use indexed access rather than table scans.

```
select authorinfo from authors
where xmlexists('$a/author[phone=4084511234]' passing
authorinfo as "a");

select authorinfo from authors
where xmlexists('$a/author[phone/areacode=408]' passing
authorinfo as "a");
```

XML indexes cannot be composite key indexes like multi-column relational indexes. That is, you cannot define a single index on two or more XML patterns. However, you can sometimes mimic a composite index if your elements are appropriately nested. For example, the index **phoneidx** above acts much like a composite index on /phone/areacode and /phone/number.

## Special cases where XML indexes cannot be used

### Special cases with XMLQUERY and XMLEXISTS

All of the discussed guidelines for XML index eligibility apply to both XQuery and SQL/XML queries. Additionally, there are some specific considerations for the SQL/XML functions XMLQUERY and XMLEXISTS.

If you use XML predicates in the XMLQUERY function in the `select` clause of an SQL statement, these predicates do not eliminate any rows from the result set and therefore cannot use an index. They only apply to one document at a time and may return a (possibly empty) fragment of a document. Thus, you should place any document- and row-filtering predicates into an XMLEXISTS predicate in the `where` clause of your SQL/XML statement.

When you express predicates in XMLEXISTS, make sure that you use square

brackets such as in `$a/author[phone=4084511234]` rather than `$a/author/phone=4084511234`. The latter of these two predicates is a Boolean predicate that returns "false" if the phone element does not have the desired value. Since XMLEXISTS truly checks for the existence of a value, even the existence of the value "false" satisfies XMLEXISTS so that any document qualifies for the result set. If you use square brackets, then the XPath expression evaluates to the empty sequence that fails the existence test and eliminates the corresponding row (if the document doesn't have the desired phone number).

For more detailed examples of these XMLQUERY and XMLEXISTS semantics, refer to "[15 Best Practices for pureXML Performance in DB2 9](#)" (developerWorks, October 2006).

### Let and return clauses

Be aware that predicates in XQuery `let` and `return` clauses do not filter result sets. Therefore, they cannot be expected to use indexes if element construction is involved. The next two queries cannot use an index because an element "phone408" needs to be returned for *every* author, even if it is an empty element for authors outside the 408 area code.

```
XQUERY for $a in db2-fn:xmlcolumn("AUTHORS.AUTHORINFO")/author
let $p := $a/phone[areacode="408"]//text()
return <phone408>{$p}</phone408>

XQUERY for $a in db2-fn:xmlcolumn("AUTHORS.AUTHORINFO")/author
return <phone408>{$a/phone[areacode="408"]//text()}</phone408>
```

### Parent steps

DB2 9 also does not use an index for predicates that occur under a parent step (".."), such as the predicate on "price" in the following two queries:

```
XQUERY for $b in
db2-fn:xmlcolumn("BOOKS.BOOKINFO")/book/title[..]/price < 10]
return $b

XQUERY for $b in db2-fn:xmlcolumn("BOOKS.BOOKINFO")/book/title
where $b/../../price < 10
return $b
```

This is not a significant limitation because you can always express these predicates without the parent axis:

```
XQUERY for $b in db2-fn:xmlcolumn("BOOKS.BOOKINFO")/book[price
< 10]/title
return $b

XQUERY for $b in db2-fn:xmlcolumn("BOOKS.BOOKINFO")/book
```

```
where $b/price < 10
return $b/title
```

## A special case with the double-slash (//)

Another case to watch out for is when you use predicates with the descendant or self axis, commonly abbreviated by //. Assume you want to find books for the author with ID 129. If author ID attributes occur at multiple levels, or if you are unsure at which level or under which element the ID attributes are located, you may write the following unfortunate query:

### WRONG!

```
select bookinfo
from books
where XMLEXISTS('$b/book/authors[@id = 129]'
                passing bookinfo as "b")
```

The intention of this query is to check ID attributes anywhere in or under the "authors" element. However, a slash or a double slash ( / or // ) at the front of a predicate in square brackets does not have a context and therefore refers to the root of the document. Thus, the query would return the following document as a result, which was not intended.

```
<book id="129">
  <authors>
    <author id="47">John Doe</author>
    <author id="58">Peter Pan</author>
  </authors>
  <title>Database Systems</title>
  <price>29</price>
</book>
```

To avoid this you need to add a dot (self axis) to indicate that you want to apply the descendant or self axis (//) from the "authors" element downward in the document tree.

### RIGHT!

```
select bookinfo
from books
where XMLEXISTS('$b/book/authors[.//@id = 129]'
                passing bookinfo as "b")
```

This also allows DB2 to use an index defined on /book//@id or //@id. No index is used if the dot is missing.

You can find further examples of how XQuery and SQL/XML language semantics

effect index eligibility in the article "[On the Path to Efficient XML Queries](#)."

## Summary

XML indexes are critical for high XML query performance but wildcards, namespaces, data types, joins, text nodes, and other semantic aspects of XML queries determine whether a certain index can or cannot be used. Some attention is required to make sure that XML index definitions and query predicates are compatible. This article presented a set of guidelines and examples to show how XML indexes are used to avoid table scans and provide high query performance. The most important guidelines are summarized in the downloadable cheat sheet.

## Acknowledgments

For their help with this paper, I'd like to thank Andrey Balmin, Kevin Beyer, Christina Lee, Henrik Loeser, Fatma Ozcan, Bryan Patterson, Vitor Rodrigues, Marcus Roy, and Cindy Saracco.

## Downloads

Description	Name	Size	Download method
Cheat Sheet	cheat_sheet.pdf	56KB	<a href="#">HTTP</a>
DB2 XML Index Exploitation - DDLandQueries.txt	DDLandQueries.txt	8KB	<a href="#">HTTP</a>

[Information about download methods](#)

# Resources

## Learn

- Refer to the [DB2 pureXML Cookbook](#) (IBM Press) for comprehensive coverage of pureXML on all supported platforms: DB2 9.x for Linux, UNIX, and Windows and DB2 9 for z/OS.
- Learn how to [Enhance business insight and scalability of XML data with new DB2 9.7 pureXML features](#).
- See [Query DB2 XML Data with SQL](#) to learn how to query data stored in XML columns using SQL and SQL/XML.
- Read the article [Query DB2 XML data with XQuery](#) to learn how to use the XQuery language in DB2.
- Check out [pureXML in DB2 9: Which way to query your XML data?](#) for more details on querying XML data in DB2 and guidelines for choosing the right approach for your needs.
- Look at the [15 Best Practices for pureXML Performance in DB2 9](#) for some XML-specific performance tips.
- Stay [On the Path to Efficient XML Queries](#) to avoid common pitfalls with XQuery, SQL/XML, and XML indexes.
- Discover the versatile XMLTABLE function in [XMLTABLE by Example](#).
- Learn how to [Update XML in DB2 9.5](#) with the XQuery Update Facility.
- Browse the [DB2 pureXML Wiki](#) for technical articles, solutions, demos, success stories, and many other resources around DB2 pureXML.
- Check out the latest [XML benchmark results](#).

## Get products and technologies

- Get hands-on with DB2 pureXML and download [DB2 Express-C](#), which is free to download, deploy, and distribute.
- Build your next development project with [IBM trial software](#), available for download directly from developerWorks.

## Discuss

- [Participate in the discussion forum for this content](#).
- Participate in the [DB2 pureXML discussion forum](#) where you can ask and answer pureXML-related questions.

## About the author

### Matthias Nicola

Matthias Nicola is the technical lead for XML database performance at IBM Silicon Valley Lab. His work focuses on all aspects of XML performance in DB2, including XQuery, SQL/XML, and all native XML features in DB2. Matthias works closely with the DB2 XML development teams as well as with customers and business partners who are using XML, assisting them in the design, implementation, and optimization of XML solutions. Before joining IBM, Matthias worked on data warehousing performance for Informix Software. He received his doctorate in computer science in 1999 from the Technical University of Aachen, Germany.