

X-Binder: Path Combining System of XML Documents Based on RDBMS

Bum-Suk Lee, Byung-Yeon Hwang*

Department of Computer Engineering, The Catholic University of Korea
{bslee, byhwang}@catholic.ac.kr

Abstract. With the increasing use of XML, considerable research is being conducted on the XML document management systems for more efficient storage and searching of XML documents. Depending on the base systems, these researches can be classified into object-oriented DBMS (OODBMS) and relational DBMS (RDBMS). OODBMS-based systems are better suited to reflect the structure of XML-documents than RDBMS-based ones. However, using an XML parser to map the contents of documents to relational tables is a better way to construct a stable and effective XML document management system. The proposed X-Binder system uses an RDBMS-based inverted index; this guarantees high searching speed but wastes considerable storage space. To avoid this, the proposed system incorporates a path combining module agent that combines paths with sibling relations, and stores them in a single row. Performance evaluation revealed that the proposed system reduces storage wastage and search time.

1 Introduction

The Extensible Markup Language (XML) [1] is designed to maintain, transfer, and process general information on the Internet by a simple method. With the increasing use of XML, the need to conduct research in various XML-related fields is also increasing. In particular, much of the recent research is related to the efficient storage and retrieval of documents that use the XML format [2].

This research is divided on the basis of base systems into object-oriented DBMS (OODBMS) [3–6] and relational DBMS (RDBMS) [7–9]. OODBMS is suitable for XML's concept; however, the use of RDBMS provides greater stability to an XML document management system. This method requires a process for mapping the structure and content of an XML document to a relational table using an XML parser. Therefore, the number of researches involving RDBMS is gradually increasing.

The XRel [7] system was developed to exploit the existing relational systems for XML retrieval. This system suggests a relational mapping scheme and path searching method; however, it does not guarantee a high search speed, and the result of the performance test shows defects in partial match and complex queries. Furthermore, the size of its path table increases continuously because it stores all the paths.

In this paper, we suggest an RDBMS-based X-Binder system to solve these problems. The proposed system includes a path combining module as an agent and query translation module. The path combining module stores the sibling nodes having the same ancestor path in a single row and thereby prevents a rapid increase in the size of the path table. The system also includes a query translation module that converts a user query into an SQL format. The module utilizes XPath's linear path expressions (LPE) [10] to retrieve XML documents. The linear path expression can be classified as a full match, partial match, and complex query. These three query types are converted to SQL queries by the query translation module.

The X-Binder system applies an inverted indexing technique [11–12] to guarantee fast performance with respect to the search speed of a user query. An inverted index is easy to implement and has the merit of speed. On the other hand, this technique wastes storage space. The system maintains an index table of various items with the size of which could be up to three times the size of the original data file. The path combining module of X-Binder exploits the existing inverted indexing technique and avoids its defect.

The system showed a reduction in the number of rows of the stored path tables when the path combining module was used. It also exhibited a high performance in terms of the path searching speed in comparative tests. The query translation module utilizes XPath's LPE.

This paper is organized as follows: Section 2 introduces the XML query model used in X-Binder. In Section 3, we describe the system architecture and the path combining and query translation modules. Section 4 presents the results of our performance experiments. Finally, in Section 5, we discuss our conclusions.

2 Related Work

In this section, we define the XML query model. Fig. 1 shows an example of an XML document that is used to explain the proposed system.

The primary XML query languages—XPath [10], XQuery [13], and Quilt [14]—use a path expression to define queries regarding an XML tree. Both LPE and branching path expressions (BPE) used in this study are the most widely used path expressions in the field of XML research. *PathFinder(p)*, the path query function, is given path p as input, which is expressed as an LPE; it, in turn, outputs the documents and text values associated with the path.

```
<movie title="Old boy" year="2003" country="Korea">
  <director>
    <fullName>Chan-wook Park</fullName><nationality>Korean</nationality></director>
  <cast>
    <players>
      <player><role>Dae-su Oh</role><name>Min-sik Choi</name></player>
      <player><role>Mi-do</role><name>Hye-jeong Kang</name></player></players></cast>
  <genre>Action, Mystery, Thriller</genre>
  <comments>
    <comment>Old Boy is definitely my favorite movie ...</comment>
    <comment>I am a big fan of Asian cinema and ...</comment></comments>
</movie>
```

Fig. 1. Example of an XML document

Definition 1. *PathFinder(p)*, a path query function, is given path p as input and it returns the set of documents and text values associated with the path.

$PathFinder(p) = \{(document, text) \mid p \text{ is used in } document \text{ and } text \text{ associated with the } p\}$

Q1 and Q2 in Fig. 2 are two basic types of LPE. The symbol “/” in Q1 represents a parent-child relationship in the path. This type of a query is referred to as a full match query. The symbol “//” in Q2, indicates an ancestor-descendant relationship, and it represents a partial match query.

| | |
|-------|----------------------------------|
| Q1 :: | /movie/director/fullname |
| Q2 :: | /movie//player/role |
| Q3 :: | /movie[@year='2003']/player/name |

Fig. 2. Linear path expression and complex query

Definition 2. A full match query is expressed as $/l_0/l_1/l_2\dots/l_n$, where l_0 is the root of the document and l_i ($i = 1, 2, \dots, n$) is the i th label of the path. The relationship between l_{i-1} and l_i is represented by “/”; this indicates that the path consists of a parent-child relationship only. l_n is the end node.

Definition 3. A partial match query is expressed as $/l_0/l_1//l_2\dots/l_n$. The “//” symbol is included in this query at least once. The “/” symbol can also be included. This query is a path expression that includes an ancestor-descendant relationship in it.

Further, we define a complex query as a combination of both LPE and a content query. Q3 in Fig. 2 shows an example of a complex query.

Definition 4. A complex query is expressed as $/l_0[l_1 = 'c_1']/l_2\dots/l_n$. In this query, the contents of l_0/l_1 include c_1 , and it selects documents with the path $l_0/l_2\dots/l_n$. A complex query uses both “/” (full match query) and “//” (partial match query).

3 An Agent System for Combining the Paths of XML Documents

3.1 System Architecture

The system architecture of X-Binder involves a path combining module as an agent for storing XML documents efficiently, a query translation module for transforming a linear path query into an SQL query to search the RDBMS, and an inverted index management module to guarantee fast retrieval. The inverted index management module operates such that an ordered status of the index table is maintained.

Fig. 3 shows the system architecture of X-Binder. An XML document is provided as input to the database through the user interface (UI), and it passes through the path combining and inverted index management modules. When a linear path query is provided as input to search a stored XML document, the query is translated into an SQL query by the query translation module. After this process, the SQL query can be used

to search the database, and the system displays the obtained search results through the UI.

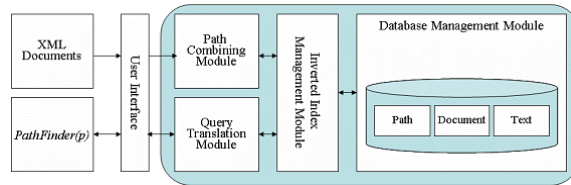


Fig. 3. System architecture of X-Binder

To use the path combining and inverted indexing techniques, we designed three relational table schemes with Document, Path, and Text tables, as shown in Fig. 4. In this database, docID, pathID, and textID are the identifiers for documents, paths, and text values, respectively. The main difference between X-Binder from XRel is that in XRel, the stored paths are combined with their siblings. XRel uses the LIKE operation to label the path match retrieval. The demerit of this method is that it results in an increase in the search time; this is because it uses a string match even when there is excessive storage data. X-Binder decreases the number of rows of a path table to be searched by combining paths, thereby providing efficiency in terms of storage space and search speed.

| | |
|----------|---------------------------------------|
| Document | docID, docName |
| Path | pathID, CombinedPathexp |
| Text | textID, docID, pathID, endPath, value |

Fig. 4. Table structure of X-Binder

Fig. 5 shows an example of a database that stores the XML document shown in Fig. 1. It stores the combined paths in the path table mentioned above. The combined paths use “#” instead of “/” in the Path table, similar to XRel. The three tables serve as an inverted index relative to the other tables.

| Text | | | | |
|--------|-------|--------|-------------|---|
| textID | docID | pathID | endPath | value |
| 1 | 1 | 1 | fullname | Chan-wook Park |
| 2 | 1 | 1 | nationality | Korean |
| 3 | 1 | 2 | role | Dae-su Oh |
| 4 | 1 | 2 | name | Min-sik Choi |
| 5 | 1 | 2 | role | Mi-do |
| 6 | 1 | 2 | name | Hye-jeong Kang |
| 7 | 1 | 3 | genre | Action, Mystery, Thriller |
| 8 | 1 | 3 | @title | Old boy |
| 9 | 1 | 3 | @year | 2003 |
| 10 | 1 | 3 | @country | Korea |
| 11 | 1 | 1 | comment | Old Boy is definitely my favorite movie ... |
| 12 | 1 | 2 | comment | I am a big fan of Asian cinema and ... |

| Document | |
|----------|---------------------|
| docID | docName |
| 1 | dataset\imdb\01.xml |
| 2 | dataset\imdb\02.xml |
| ... | ... |
| ... | ... |
| 10 | dataset\imdb\10.xml |

| Path | |
|--------|--|
| pathID | CombinedPathexp |
| 1 | #/movie#/director#/(fullname#/#/nationality#) |
| 2 | #/movie#/cast#/players#/player#/(role#/#/name#) |
| 3 | #/movie#/(genre#/#/@title#/#/@year#/#/@country#) |
| 4 | #/movie#/comments#/(comment#) |

Fig. 5. Example of a stored table of X-Binder

3.2 Path Combining Module

Few studies—1-Index [15], A(k)-Index [6], and DataGuides [16]—have investigated the construction of a small and fast index by combining similar paths, but these techniques are based on graph indexing techniques. These procedures combine nodes, not only the end nodes but also the middle nodes of a path. Therefore, they are not relevant to our research, which is based on RDBMS.

In this section, we describe a path combining module of X-Binder. The proposed method extracts a complete path from the root to the end node with a value, and it regards the paths with the same ancestor path (except the end nodes) as similar paths. The module combines these paths and stores them in a row in a relational table. If there exist attributes and elements with the same parent, the module adds an attribute identifier “@” to the head of an attribute name and recognizes the nodes as sibling nodes.

Definition 5. Combining the paths of XML documents refers to the binding of all the sibling paths (sibling paths: $p_1, p_2, p_3, \dots, p_n$) and storing them in the rows of a relational table.

PathCombine(sibling_paths) = {(CombinedPathexp) | sibling_paths are several sibling path inputs. CombinedPathexp represents a stored path expression that is a combination of the paths provided as inputs.}

For example, when the *PathCombine(sibling_paths)* operation is executed for the XML document shown in Fig. 1, ‘/movie/director/fullname’ and ‘/movie/director/nationality’ have the same relative ancestor paths and only their end nodes differ; thus, they have a sibling node relationship. Further, ‘/movie/genre’, ‘/movie/@title’, ‘/movie/@year’, and ‘/movie/@country’ are siblings, and also ‘/movie/cast/players/player/role’ and ‘/movie/cast/players/player/name’ are siblings. Lastly, the path ‘/movie/comments/comment’ is stored in the table by itself because it does not have any sibling nodes.

By combining and storing sibling paths using this method, four path expressions are generated, as shown in the path table in Fig. 5. This result is comparable to the path table of the XRel system, which extracts fourteen path expressions. In Fig. 5, the “|” symbol is used to distinguish the end nodes from the combined paths.

Fig. 6 shows the algorithm of this method. We disregard the significance of the “/#” symbol in this explanation. The algorithm requires four inputs: fullPath, endPath, queryFullPath, and queryPrePath. For instance, if there exists a path ‘/aa/bb/cc’ to be used for storage, ‘/aa/bb(cc)’, ‘cc’, ‘/aa/bb/(%cc%)’, and ‘/aa/bb/(%)’ could be the four inputs. The first parts of two of them are the inputted values to be stored, and the others are required for searching the database.

The algorithm can process for the following three cases: (1) queryFullPath already exists in the Path table; (2) queryFullPath does not exist, but queryPrePath does; or (3) both queryFullPath and queryPrePath do not exist. In the first case, the algorithm merely returns the pathID of the row of the existing path. In the second case, a path similar to ‘/aa/bb/(dd|ee|ff)’ exists; therefore, the algorithm modifies the Path table to

‘/aa/bb/(dd|ee|ff|cc)’ and returns the pathID. Lastly, in the third case, a new input path is provided that does not exist in the table. Therefore, the algorithm stores this new input path in the form of ‘/aa/bb/(cc)’; the parentheses at the end node are used for distinguishing it. This method separates ‘/aa/bb/cc/gg/(hh|ii)’ from ‘/aa/bb/cc’.

```

Input : fullPath, endPath, queryFullPath, queryPrePath
Output : pathID
Method :
1  resultQFP = getStoredPath(queryFullPath);
2  if(resultQFP != null){
3    Return pathID;
4  }else{
5    resultQPP = getStoredPath(queryPrePath);
6    if(resultQPP != null{
7      CPathexp = combinedPath(resultQPP, endPath);
8      updatePath(CPathexp);
9      return pathID;
10   }else{
11     insertPath(fullPath);
12     return pathID;
13   }
14 }

```

Fig. 6. Path combining algorithm

3.3 Query Translation Module

It is impossible to directly search a relational database with an LPE to execute the *PathFinder(p)* function. Accordingly, to obtain information regarding the XML documents that are stored in an RDBMS, a query translation process that converts an LPE to an SQL query is required.

3.3.1 Full Match Query

A full match query is a type of linear path query that traverses from the root to the end node. Assuming that a query ‘/movie/cast/players/player/role’ is provided as input, it has to be converted to an SQL query. To use the structure of the Text table, a query might be separated by using an end node ‘role’ and an ancestor path. Fig. 7 shows the SQL query of the input. The third line in Fig. 7 restricts the endPath of the Text table to ‘role’; thus, information regarding ‘name’, which is stored with ‘role’, might be ignored.

```

1  SELECT  d1.docName, t1.value
2  FROM    Document d1, Path p1, Text t1
3  WHERE   t1.endPath = 'role'
4  AND     p1.CombinedPathexp LIKE '/movie#/cast#/players#/player#/(%'
5  AND     p1.pathID = t1.pathID
6  AND     t1.docID = d1.docID

```

Fig. 7. An SQL query for ‘/movie/cast/players/player/role’

3.3.2 Partial Match Query

A partial match query contains more than one ‘/’ symbol, which indicates an ancestor–descendant relationship, and it might not be suffixed with the root. We can easily convert this type of query to an SQL query by including conditions such as JOIN and LIKE operations. For instance, in the query ‘/movie//cast’, the ‘/’ symbol will select

all paths that have ‘cast’ at a lower position in documents with the root ‘movie’. It is easy to detect results by using the string match query method. In order to make the abovementioned query consistent with the format of X-Binder’s database, “/” and “//” have to be changed to “#” and “#%/”. Fig. 8 shows the SQL query for ‘/movie//cast’.

```

1 SELECT d1.docName, t1.value
2 FROM Document d1, Path p1, Text t1
3 WHERE p1.pathID = t1.pathID
4 AND t1.docID = d1.docID
5 AND (p1.CombinedPathexp LIKE '#/movie#%/cast#%('
6 OR (p1.CombinedPathexp LIKE '#/movie#%(/cast#%'
7 AND t1.endPath = 'cast'))

```

Fig. 8. An SQL query for ‘/movie//cast’

In Fig. 8, the condition of the location of ‘cast’ and ‘movie’ is restricted by the WHERE clause. The condition stated on the fifth line is that ‘cast’ should not be an end node, and that on the sixth line states that ‘cast’ should be an end node. Moreover, it checks again whether endPath of the Text table is ‘cast’ or not. For the query ‘//movie//cast’ wherein ‘movie’ may or may not exist as the root, the module will insert “%” before the LIKE operation in lines five and six. Therefore, the final forms are ‘%/movie#%/cast#%(%’ and ‘%/movie#%(/cast#%’.

3.3.3 Complex Query

A complex query is a combination of a linear path query and a content query. The query ‘/movie[@year=‘2003’]//player/name’ selects a set of documents with the path ‘/movie//player/name’ and an attribute value of ‘year’ in which the low-level node of ‘movie’ is 2003. This query can be separated into two parts—‘/movie/@year=‘2003’ and ‘/movie//player/name’. To consider this, the query generating module of X-Binder converts this query to one similar to that shown in Fig. 9. The second line creates table t1 for the first part, and the fourth line creates table p1 for the remaining part.

```

1 SELECT d1.docName
2 FROM (SELECT * FROM Text
3 WHERE endPath = '@year' and value = '2003') t1,
4 (SELECT * FROM Path
5 WHERE CombinedPathexp LIKE '#/movie#%/player#(%/name#%'
6 Document d1
7 WHERE (t1.docID = d1.docID)

```

Fig. 9. SQL query for ‘/movie[@year=‘2003’]//player/name’

4 Performance Evaluation

The result of the performance test of X-Binder is discussed in this section. The test environment comprised the following: MS Windows XP Professional was used as the operating system and MS-SQL was used as the base DBMS. X-Binder was developed using JAVA 1.4.2.09 and JDOM 1.0 [17] was used for XML parsing. There are two aspects to the performance test. First, we compared the storage spaces of XRel that represents the RDBMS based system and X-Binder that represents our suggestion. Due to the differing table structures, we considered the designed Path tables to have

similar structures. Second, we compared the search speeds of the *PathFinder(p)* function in the two systems.

4.1 Comparison of Storage Space

As datasets for the experiment, we used the IMDB database in XML format [18], a Reuters news file with NewsML [19], and a random dataset from the ReGet program [20] that collects documents on the Internet. The maximum depths of each dataset were 6, 9, and 14, and the numbers of documents were 100, 250, and 300, respectively.

We counted the number of rows in the Path tables of both XRel and X-Binder. Fig. 10 shows the result: the number of paths stored in the X-binder database was lesser than that stored in the XRel one. This is a reasonable result because XRel stores all the paths from XML documents separately. However, X-Binder stores the sibling paths in a row. Next, we compared the database sizes. The size of the database in the proposed system was also smaller than that of XRel.

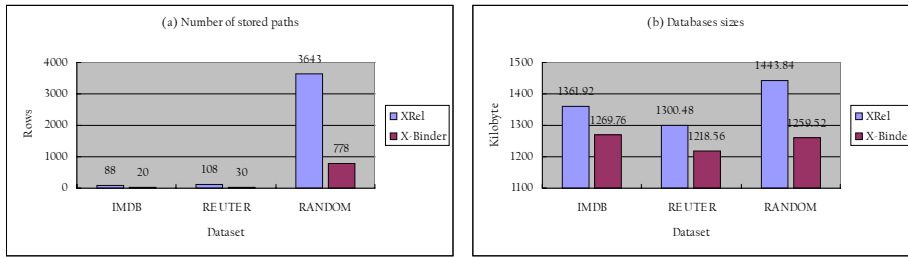


Fig. 10. Comparison of the database sizes and the numbers of stored paths

4.2 Comparison of Search Speeds of *PathFinder(p)*

Fig. 11 shows the list of queries that were created to evaluate the speed test of the *PathFinder(p)* function. Q1 and Q2 were full match queries with different lengths. Q3 and Q4 were partial match queries, and they included the “//” symbol once or twice each. Lastly, Q5 and Q6 were complex queries comprising a full match query and a partial match query.

| | | |
|----|---|-------------------------------|
| Q1 | /seller/name | Full match query(Short) |
| Q2 | /movie/cast/player/filmographies/filmography/work | Full match query(Long) |
| Q3 | //topic/topicname | Partial match query(one '//') |
| Q4 | //news//topicname | Partial match query(two '//') |
| Q5 | /members/member/id/name/[first='Lee'] | Complex query |
| Q6 | /members/member/[first='Seo'] | Complex query with '//' |

Fig. 11. Queries for performance test

The result of the performance test for *PathFinder(p)* is shown in Fig. 12. We found that the X-Binder system exhibited a better performance than the existing XRel system. In particular, while executing a complex query, the search time was decreased by

25–33%. This was caused by the decrease in the number of rows of the Path table, which in turn decreased the number of rows that were created after the JOIN operation.

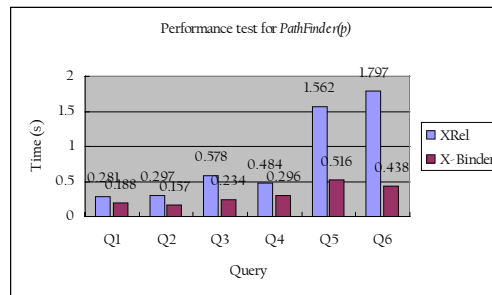


Fig. 12. Comparison of the search speeds of *PathFinder(p)*

5 Conclusion

An increasing amount of research is being conducted on various RDBMS-based XML document management systems. The systems that use an inverted index typically guarantee fast retrieval, and their performance results are superior. However, an inverted index composes a new index that considers the values of the special fields of a database table, and this results in the wastage of storage space.

To solve this problem, we suggest an X-Binder system that has a path combining module based on RDBMS. The proposed system combines sibling nodes and stores them in a single row in a relational table. This method decreases the number of rows of a Path table, and therefore utilizes less storage space.

We compared the proposed system and XRel in terms of the storage space and search speed of *PathFinder(p)* to prove that our system exhibits a better performance. The results revealed that the number of rows of stored data decreased by 20–25%. An experiment for comparing the sizes of the databases showed an overall decrease in size. The X-Binder system guarantees high search speed and better performance, especially for complex queries.

In future studies, considering an inverted index system using a posting list might improve performance. In addition, in order to generate an efficient query, the calculation of the search cost using SQL might yield interesting results.

References

1. W3C: Extensible Markup Language (XML) Version 1.0 (Second Edition), <http://www.w3c.org/TR/REC-xml> (2000)
2. Ceri S., Fraternali P., Paraboschi S.: XML: Current Developments and Future Challenges for the Database Community. Proc. of the 7th Int'l Conf. on EDBT (2000) 3–17

3. McHugh J., Abiteboul S., Goldman R., Quass D., Widom J.: Lore: A Database Management System for Semistructured Data. *ACM SIGMOD Record*, Vol. 26, No. 3 (1997) 54–66
4. Cooper B. F., Sample N., Franklin M. J., Hjalton G. R., Shadmon M.: A Fast Index for Semistructured Data. *Proc. of the 27th Int'l Conf. on VLDB, Rome, Italy (2001)* 341–350
5. Chung C., Min J., Shim K.: APEX: An Adaptive Path Index for XML Data. *Proc. of the Int'l Conf. on ACM SIGMOD, Madison, Wisconsin (2002)* 121–132
6. Kaushik R., Shenoy P., Bohannon P., Gudes E.: Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. *Proc. of the 18th IEEE Int'l Conf. on Data Engineering (2002)* 129–140
7. Yoshikawa M., Amagasa T., Shimura T., Uemura S.: XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM ToIT*, Vol. 1, No. 1 (2001) 110–141
8. Jiang H., Lu H., Wang W., Yu J. X.: Path Materialization Revisited: An Efficient Storage Model for XML Data. *Proc. of the 13th Australasian Database Conf., Melbourne, Australia (2002)* 85–94
9. Jiang H., Lu H., Wang W., Yu J. X.: XParent: An Efficient RDBMS-Based XML Database System. *Proc. of the 18th Int'l Conf. on Data Engineering, San Jose, California (2002)* 335–336
10. Clark J., DeRose S.: XML Path Language (XPath) Version 1.0, W3C Recommendation (1999)
11. Zhang C., Naughton J., Dewitt D., Luo Q., Lohman G.: On Supporting Containment Queries in Relational Database Management Systems. *ACM SIGMOD Record*, Vol. 30, No. 2 (2001) 425–436
12. Florescu D., Kossmann D., Manolescu I.: Integrating Keyword Search into XML Query Processing. *Proc. of the 9th Int'l WWW Conf. on Computer Networks (2000)* 119–135
13. W3C: XQuery 1.0: An XML Query Language. <http://www.w3.org/TR/2005/WD-xquery-20050915/> (2005)
14. Chamberlin D., Robie J., Florescu D.: Quilt: An XML Query Language for Heterogeneous Data Sources. *Proc. of the 3rd Int'l Workshop on ACM WebDB (2000)*
15. Milo T., Suciu D.: Index Structure for Path Expressions. *Proc. of the 7th Int'l Conf. on Database Theory (1999)*
16. Goldman R., Widom J.: Approximate DataGuides. *Proc. of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats, Jerusalem, Israel (1999)* 436–445
17. <http://www.jdom.org>
18. http://us.imdb.com/top_250_films
19. <http://about.reuters.com/newsml>
20. <http://deluxe.reget.com/en/>