EFFICIENTLY QUERYING THE INDEXED COMPRESSED XML DATA (IQX)

Radha Senthilkumar[1] and Kannan Arputharaj[2]

[1]Department of Information Technology, MIT, Anna University, chennai, India
radhasenthil@annauniv.edu
[2]Department of Information Science and Technology, CEG, Anna University, Chennai India
kannan@annauniv.edu

ABSTRACT

Extensible Mark-up Language was designed to carry data which provides a platform to define own tags. XML documents are immense in nature. As a result there has been an ever growing need for developing an efficient storage structure and high-performance techniques to query efficiently. QUICX (Query and Update Support for Indexed and Compressed XML) is the compact storage structure which gives highest compression ratio than many of the queriable compressors available today. The data are compressed using LZW approach, and stored compactly. Indexing is done for the data stored in the containers, there by further increasing the compression ratio. These also reduce the time for querying the storage structure. Index files are created only for highly redundant files which is larger in size. IQCX support simple, aggregate, conditional, compound, nested and correlation predicate queries. All queries except simple query use the index files. The reduced execution time proves the efficiency of the indexing technique.

KEYWORDS

XML, XPath, Indexing, Query processing.

1. INTRODUCTION

XML is flexible in managing data. Its design goals mainly focus on simplicity, and increasing the usability over the Internet. XML documents are huge in size thus an efficient storage structure is required. In the QUICX [9] structure, the XML document which is given as input is parsed and the details are stored in three different structures namely metatable, metadata and containers. Meta table stores the tag id, name of the node, level id, parent id and the file id. Meta data stores the tag id of the nodes in the same order as it is in the XML document. The actual data within the nodes is stored in containers and 50 records in each. This is how the entire XML document is divided and stored. The proposed indexing technique is based on the redundant records found in the containers. If more redundant records are present in a container then it is chosen for indexing. The information available in the Meta data file is sometimes used for indexing. Once the file is indexed, the original container is removed and only the index is maintained. This is because the index file contains the data as well as the location where it is present in the original container. Hence, there will be no loss of data and the indexing technique leads to the deletion of some of the huge containers thereby reducing the memory wasted in storing them and increasing the compression ratio. Querying the XML data using index is carried. This proves that there is a decrease in the overall execution time though considerable amount of time is spent in creating the index file.

DOI: 10.5121/ijdms.2011.3312

2. RELATED WORK

Indexing structure, extended inverted index technique [1], used in information retrieval is proposed for processing queries efficiently. Four types of indexes are defined which are stored in relational RDBMS. One among them stores the paths and gives it an id. And one of the others stores the value of the path. Start and End offset of the nodes is also stored so that searching range is limited and query performance is enhanced. More memory space is required to store the details. But then, once the details get stored, querying becomes easy with simple join operations performed. IQCX technique does not store all these details. In IQCX the data is simply stored in compressed form thus reducing the memory required for storage.

The design [10], have two compressed data structures for the full-text indexing problem that support efficient substring searches using roughly the space required for storing the text in compressed form are proposed. The first compressed data structure retrieves the occurrences of the pattern within the text. The second compressed data structure provides optimal output-sensitive query time. This second data structure builds upon the first one. However IQCX indexing technique still produces an optimal algorithm for querying. The demerits of using certain techniques and optimized algorithm to provide good results are discussed.

Two techniques are introduced [9] for efficient memory resident path indexes. The first simple approach combines inverted lists, selectivity estimation, hit expansion, brute force search. The second uses suffix trees with additional statistics with multiple entry points into the query. The entry points are partially evaluated in an order based on estimated cost until one of them is complete. But the brute force technique might take much time in finding the actual data required. In this work brute force technique is not used to find out the actual data required. The index file is used in locating the records of interest and only those data are decompressed and displayed.

A new methodology is proposed [6] with multi dimensional approach for retrieving attributes. The method proposed enables indexing of a collection of attributes to facilitate approximate retrieval using edit distance as an approximate match predicate for strings and numeric distance for numeric attributes. Frequently occurring strings are given minimal length code. And infrequent strings are given comparatively greater length code. Assigning code increases the memory requirement to store it. In IQCX, only the data in the XML file eliminating the tag names are stored and not the extra information about the tags. Thus overall memory required to store the data is reduced.

DRXQP[3] technique stores the encoded value of element paths, attributes, contents of the element paths and attributes, and XML processing instructions in a dynamic relational structure termed as Multi-XML Data-Structure (MXDS). The encoded values are calculated based on the parent-child relationship. Encoding and query processing becomes complex when the size of the original XML document increase. IQCX do not generate any code from the input data and do not maintain tables with the calculated values. Thus memory space as well as time to execute the query gets reduced. Xlight[15] is used to store all types of XML documents and is composed of five tables. Document, Path, Data, Ancestor, Attribute. Querying becomes complex due to merging of tables for data retrieval. IQCX approach overcomes this problem by storing data in containers; the index file generated is used to retrieve only particular records required.

The means of effectively identifying the type of target node(s) that a keyword query intends to search for is suggested in [2]. The types of condition nodes that a keyword query intends to search via is effectively inferred. Each query result is ranked. A formula to compute the confidence level of a certain node type to be a desired search for node is designed. Formulae to compute the confidence of each candidate node type as the desired search via node to model natural human intuitions. The pattern of keywords co occurrence in query is considered. However some of the irrelevant keywords may get displayed. IQCX algorithm executes the query after validating it against the Meta table contents.

IQCX select only the desired records, decompress and print them. Thus irrelevant details will not get printed.

Navigation-Free Data Instance [8] is the combination of the runtime generated path tables and token buffers of a given XQuery Q over data stream D. Token Buffers store the offset of nodes related to the query. Considerable amount of memory is wasted in storing the offsets and time is also wasted in calculating them.A compressed index for XML, called XBzipIndex[11] is designed, and the XML document is maintained in a highly compressed format, and both navigation and searching is done uncompressing only a tiny fraction of the data.

Compressing and indexing two arrays derived from the XML data is done for the data retrieval. XBzipIndex has compression ratio up to 35% which is better than the ones achievable by other tools. But then, IQCX indexing techniques, the first one using Meta data and the second one without using Meta data file contents give a better compression ratio compared to the compression ratio from XBzipIndex.

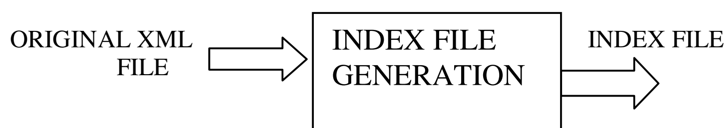
XQzip[4] supports a wide scope of XPath queries such as multiple, deeply nested predicates and aggregation. But IQCX technique proves that executing aggregate queries using the index takes lesser time when compared to executing such queries in XQzip. IQCX also supports other queries. RFX (Redundancy Free Compact Storage Structure) [13] Compact Storage is where the XML documents are stored, with no redundancy of elements. An entirely new technique is developed to process XPath queries in the RFX, exploiting the self optimized nature of Compact Storage, using Strategy List. QUICX[12] is an efficient compact storage structure that supports both query and update efficiently.

TREECHOP[5] is a queriable XML compressor that employs a homomorphic compression scheme and implements a sequential query algorithm and lazy decompression. TREECHOP, is not schema aware and implements top-down exact-match and range queries of the compressed document. TREECHOP's compression algorithm is highly efficient and the single pass scheme makes it ideal for data exchange across networks. But the compression algorithm used in QUICX compresses data efficiently and the queries executed using IQCX approach proves that the execution time is less compared to TREECHOP.

XSAQCT[14] approach compresses data and stores it and decompression is done while query processing. Simple queries executed using XSAQCT are compared with IQCX (Indexing and querying compressed XML data) technique. IQCX proves to be efficient than XSAQCT and TREECHOP techniques. A new ternary operator is developed, called Kappa-Join[7], for efficiently evaluating queries with existential quantification. A correlation predicate which can occur conjunctively and disjunctively in XML queries are decorrelated efficiently in this work. Oracle accesses every row in the inner table for each and every row in the outer table. This will definitely consume much time in the case of a nested query and this problem is overcome in this work by executing the inner query first and then using its output, the outer query is executed. Correlation predicate query which is used in IQCX approach is based on the query given in this paper.

3. OVERALL ARCHITECTURE

The overall architecture as show in Figure 1: involves loading the Meta table, dictionary file and Meta data, which is followed by query validation. Then the index of the desired record is found from the index file if necessary and the corresponding container is decompressed and retrieved.



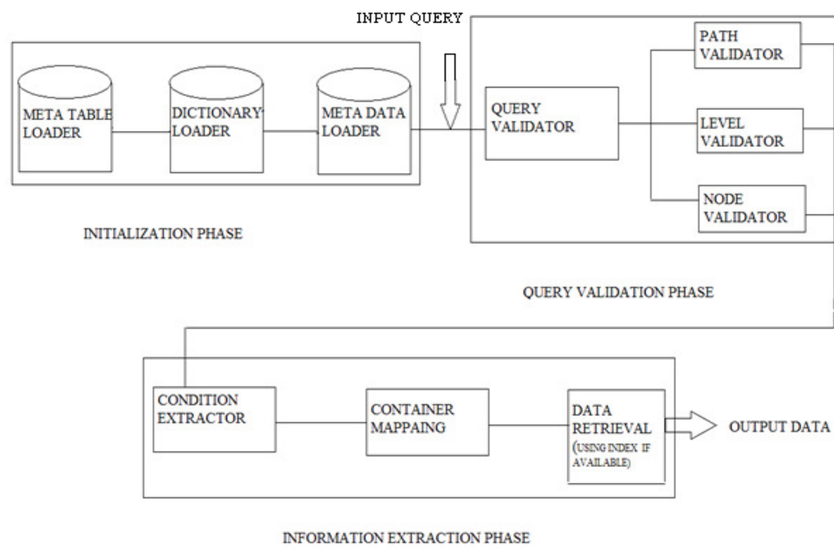


Figure 1: Architecture for indexing and querying XML data

INITIALIZATION PHASE

Once the dataset is obtained, the files with redundant records are found out. Those files are indexed and made ready to be used while querying.

- META TABLE LOADER

Initially, Meta table which is in the form of a file is loaded to validate the query by reading each and every line. Values are separated with '>' as the delimiter. Tag id is stored in the array named tag and node name is stored in the array named node and the corresponding level id and parent id arrays level, par respectively.

- DICTIONARY LOADER

A dictionary file for the entire data in the containers is generated during compression. This file is necessary during decompression of desired records. Hence it is loaded initially along with the Meta table and Meta data.

- META DATA LOADER

Meta data stores the tag id of the nodes in the same order as it is in the XML document .The tag ids are separated with ‘>’. This file is used in the information retrieval phase to find where the particular node is present.

QUERY VALIDATION PHASE

QUERY VALIDATOR

Once the initialization phase is over, Meta table is displayed in the window. User can refer this and enter their query. Query validation consists of three phases.

- ✓ Path Validation
- ✓ Level Validation
- ✓ Node Validation

PATH VALIDATOR

Path validator reads the query and finds if it is the absolute path or the relative path given to reach the particular element. If it is the absolute path, query should start with a single '/' followed by the root node. This is checked and the error is reported if any. Relative paths can have '//' which is followed by the name of the node.

LEVEL VALIDATOR

Level validator checks the parent child relationship. Starting from the first node in the query, stores the level id of each and every node traversing from left to right. Finally it checks if the successive node's level id is greater than the present. It reports the error if any.

NODE VALIDATOR

Node validator checks for errors in the node specified in the given query. If no such node exists with the given name, error is reported.

INFORMATION EXTRACTION PHASE

CONDITION EXTRACTOR

Once the query is validated, condition evaluator checks for the conditions in the query. For simple query there will not be any condition. Compound query has more than one condition in it. Aggregate query has one or more functions in it.

CONTAINER MAPPING

The index, has been found out (if available), helps us in mapping the data in the container among the list of records available. Two different files are involved.

- ✓ Compressed File
- ✓ Dictionary File

While executing a query, compressed file with the extension ".cc", Dictionary file with the extension ".dicchk" are given as input to the decompressor. Thus the query which is taken as input undergoes validation, the container to be retrieved is identified and the decompressed data is displayed.

4. CLASSIFICATION OF QUERIES

Six different types of queries with the corresponding algorithm and example are discussed here. Taking this given snapshot different types of queries are discussed.

```
<paper>
    <entry year = “2003”>
        <journal>
            <title>Secret Sharing </title>
        </journal>
    </entry>
    <entry year = “2003”>
        <conference>
            <title>XML Water Mark</title>
        </conference>
    </entry>
</paper>
```

Figure .2 A sample XML file

4.1. Simple Query

It has no condition and can be executed in less amount of time.
E.g. /paper/entry/journal/title
This query is validated, and list of all title of the journals get displayed.

4.2. Conditional Query

It has a condition that needs to be tested.
E.g. /paper[@year=”2003”]/journal/title
This query, after validation, displays the title of all the journals published in the year 2003.

4.3. Aggregate Query

Aggregate query basically involves functions like sum, count, max, min, avg.
E.g. //entry[count(journal)>50]/@year (based on the Fig. 2)
This query prints the years in which the total number of journals entered is greater than 50.

4.4. Compound Query

It has two or more conditions where conditions together decide the result.
E.g. /paper[@year >=”2003” && @year <=”2005”]/journal/title (based on the Figure.2)
This query displays the title of all the journals entered between the year 2003 and 2005.

4.5. Nested and Correlation Predicate Queries

CORRELATION PREDICATE QUERY

In a table, there can be more than one attribute and the condition given in the query can be based on the value of an attribute present in two different rows of the same table.

NESTED QUERY

It has more than one query nested in it. The inner query is said to be independent and the outer query is said to be dependant.
Example of a nested query from Shakespeare dataset
//PERSONAE/PGROUP[PERSONA=[//PERSONAE/PGROUP[GRPDESCR=”triumvirs”]/PERSONA]]/TITLE
This query can also be considered as a correlation predicate query. Thus the queries are classified and all the queries said above are supported by IQCX

5. INDEXING ARCHITECTURE

XML documents are massive in size. Redundancy is commonly found in many of the XML files. This redundancy found in data can be used to index files which enhance querying efficiency by reducing the time required to locate the record of interest. The indexing architecture shown in Figure.3 involves getting the query as input from the user. Based on the query, the index file is searched for. If it is available, then it is loaded and the corresponding record is located using the index.

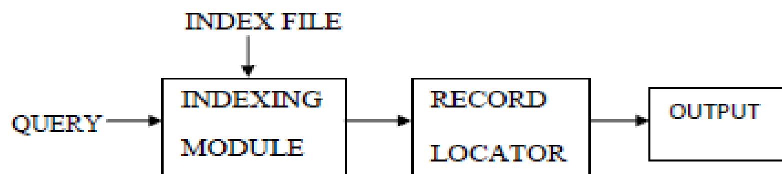


Figure. 3 Architecture for indexing XML data

The Index file is created by traversing the Meta data file once. It has the record number and the corresponding details of the record. In indexing module the query and index file is taken as input, and performs the validation. The record locator uses the indexed file and finds the particular record to retrieve; it reduces the overhead of traversing the entire container to find the desired record. The decompressor, based on the condition evaluated by the condition evaluator and based on the index, the respective records are decompressed and displayed by the decompressor.

6. CATEGORIZING FILES

To perform indexing, we need to find out the containers with redundant data. Only such containers can be indexed. If the file size is less than 1000 bytes, it can be left as such since indexing such files increases the overhead. Rest of the containers can or cannot be indexed based on the degree of redundancy. The flowchart for categorizing files is given as shown in Figure 4.

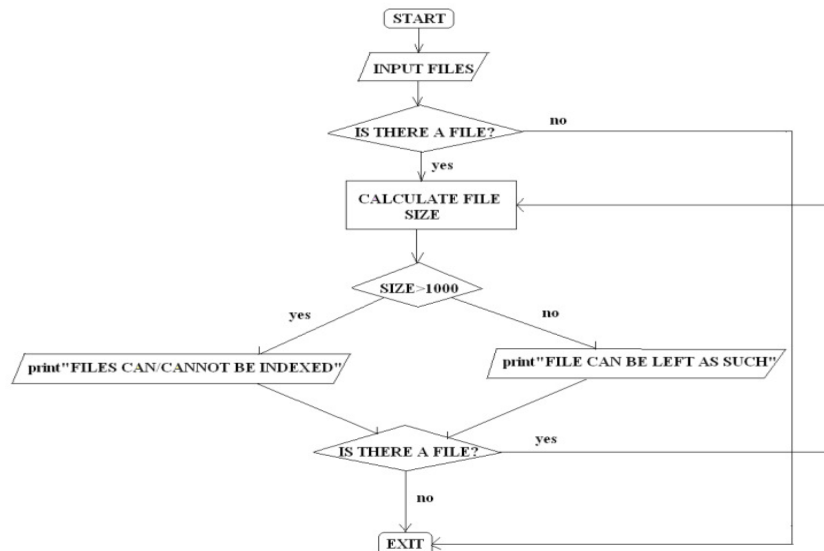


Figure 4 Flowchart for Categorizing files

The following algorithm is used to categorize files according to their size.

Procedure categorize ( )

begin

1. Input the file names.
2. Calculate each and every file’s size
3. if(filesize<1000)
4. print “File can be left as such”
5. else
6. print “File can or cannot be indexed”
7. end

end

Thus it is concluded that the files whose size is greater than 1000 bytes and which contain redundant data in them can be indexed and the rest of the files are left as such.

7. TYPES OF INDEXING

Indexing technique mainly focuses on the XML files which contain high degree of redundancy. Once the index is generated, the original file can be removed which leads to efficient utilization of memory storage. Complex queries are evaluated in much lesser time compared to the existing techniques. The index file which has the record number and the corresponding details of the record is created either using the Meta data file or without using the Meta data file, the algorithm is shown below. From this index file, desired record is found to be decompressed.

Procedure index_without_Metadata()
/* generating index file */
Begin
1. Input the file to be indexed
2. Find out the records which are different
3. Find out the position numbers where each and every record is present.
4. For each and every record present
5. Write to the index file the record followed by ‘>’
6. Append the position numbers (separated by ‘>’) where the particular record is present.
End For
End

Procedure index_speaker_with_Metadata()

/* generating index file */
Begin
1. Input the file to be indexed
2. Input the Metadata file.
3. Traverse the Metadata file and count the number of lines spoken by each speaker.
(i.e count the number of 21s after each 20)
4. For each and every speaker in the speaker container
5. Write to the index file the name of the speaker followed by ‘>’
6. For each and every set of line numbers
7. Append the starting line number followed by ‘+’ followed by total number of lines spoken at that instance – 1.
8. Append ‘>’
End For
End For
End

Thus indexing is done by these two procedures and the index file generated is saved for later use.

7.1 EXAMPLES

The way of indexing the containers using two different techniques are explained below. Examples are taken from the Shakespeare dataset. There are 58 containers (58 different tags) storing the data. Let us take some sample records from the container 20.cont which has the names of the speakers, 21.cont which has the lines spoken by each speaker and the corresponding Meta data file.

Contents of 20.cont

PHILO>CLEOPATRA>MARK ANTONY>CLEOPATRA>DUKE VINCENTIO> PHILO>

Contents of Meta data file

1>2>3>4>20>21>21>19>2>3>20>21>21>21>21>21>21>19>20>21>21>2>3>20>21>20>21>21>21>19>20>21>21>21>21>

Contents of the Index file

PHILO>1+1>15+3>
CLEOPATRA>3+5>11+0>
MARK ANTONY>9+1>
DUKE VINCENTIO>12+2>

Figure 5 Indexing Speaker Container using Meta data file

Meta data file and container file are opened. Meta data file is traversed once and the number of lines each and every speaker has spoken is found out. The speaker's name is written to the file. Then the starting line number and the line number to which he has spoken is written to the index file with a '-'(hyphen) in between. But it is found that, this technique increases the size of the index file which is greater than the original file if the original file is of huge size. Hence to optimize this technique, we find the total number of lines spoken by a speaker at a particular instance, we write the starting line number and the total number minus one to the file. These two numbers are separated by '+'.

E.g PHILO>1+1>15+3>

PHILO is the name of the speaker. First occurrence of "20" in the Meta data file represents the speaker "PHILO". Because the first name in the speaker container is "PHILO". First "1" denotes first 21 (id of line tag) which is followed by "20" in the Meta data file. Total number of lines spoken by him in the first instance is 2 (count of 21s). So the second "1" in this example represents the total number of lines – 1 i.e 2-1. Thus 1+1 as a whole represents the number of lines spoken by him at the first instance. Likewise, after 4 speakers PHILO speaks again. Thus 6th occurrence of "20" represents PHILO again. 15 represents 15th occurrence of 21. Total number of lines spoken by him, the second time is 4. So the number 3 in the example represents the total number of lines – 1 i.e 4-1. Thus 15+3 as a whole represents the number of lines spoken by him at the second instance. It is found that this technique greatly reduces the size of the index file. Now, after the index file is generated, the file 20.cont is removed and the index file is named 'speakerind.txt' and is used for querying later. This reduces the overall storage space required. The above example uses Meta data file. Now let us index the title container taking some sample records without using Meta data file.

Contents of 15.cont

ACT I>ACT II>ACT III>ACT IV>ACT V>ACT I>ACT II>ACT III>ACT IV>ACT V> ACT I>ACT II>ACT III>

Contents of the Index file

ACT I>1>6>11>
ACT II>2>7>12>
ACT III>3>8>13>
ACT IV>4>9>
ACT V>5>10>

Figure 6. Indexing Title Container without using Meta data file

The unique records present in the container are found out. For each and every unique record present, the position where the particular record is found in the entire container is identified. To the index file, the record is written and then the positions are appended to it with '>' as the delimiter. Likewise, the containers which can be indexed are found and once the index files are generated, the original files are removed. Querying can be done using the index file. Thus datasets are indexed.

7.2 OPTIMIZED INDEXING

When the dataset Lineitem was indexed, the problem is the size of the original file was smaller than the size of the index file. But indexing is done to reduce the size of the individual containers thus reducing the overall memory space required. In order to overcome this problem, we used an optimized approach to reduce the size of the index file.

ORDINARY INDEXING TECHNIQUE

The original record is written first which is followed by the positions where the particular record appears in the entire container.

E.g

0.02>1>3>5>6>7>12>14>15>17>19>25>27>28>29>30>31>…..>20390>20400>20401>20402>20403>….

The above example means that 0.02 is the original data which is present in the positions 1,3,5,6,7,12,14,15,17,19,25,27,28,29,30,31,…, 20390, 20400, 20401, 20402, 20403… It is better to write 0.02 to the file instead of storing its indexes 20390, 20400, 20401, 20402, 20403. Because, the maximum number of characters present in the original data is 4.(e.g 0.02, 0.04, 0.06, .100). But when we write 20390, 20400 etc to the file, we are writing 5 characters. Hence, the size of the index file exceeds the size of the original container.

REVISED TECHNIQUE

In order to overcome the problem stated above, we use a new approach. If the same data is present successively more than once, then the starting position number is written to which the number of times it appears successively is appended after a '+' symbol. Hence the example given in the ordinary indexing technique is modified as

0.02>1>3>5+2>12>14+1>17>19>25>27+4>…..>20390>20400+3>…

Now, instead of writing 20400, 20401, 20402, 20403 (20 characters) to the file we write only 20400+3 (7 characters) to the file. Thus, by this way, the size of the index file is reduced drastically.

8. QUERYING COMPRESSED XML DATA

8.1 QUERY EXECUTION

Querying involves loading the Meta table which is followed by query validation. Then the index of the desired record is found and the data is decompressed and retrieved. The querying architecture is shown in Figure 7.

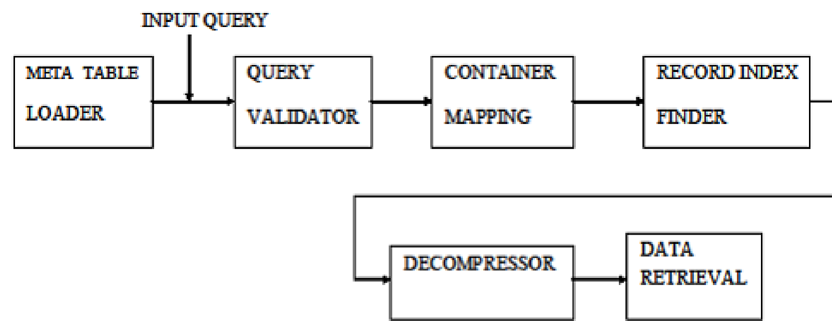


Figure 7. Architecture for querying XML data

Processing steps are

- Meta table is loaded by storing the tag id, parent id, level id and name of the node in separate buffers.
- Query is input by the user.
- Query is validated for its correctness and this validation is of three types. They are path, node and level.
- Once the query is validated, container is mapped based on the conditions in the given query.
- Record index is found, and obtain the location of the records in the container to be retrieved.
- All the modules are the same as said in the overall architecture except record index finder and decompressor.

RECORD INDEX FINDER

If the index file is available for the mapped container, then it is opened. The indexes of the records are found out based on the condition given in the query. This output is given to decompressor as input.

DECOMPRESSOR

Decompressor decompresses only the particular records identified by the indexes and the output is displayed to the user.

8.2 SIMPLE QUERY

Simple query is the one which is very easy to execute. It has no conditions in it. We need to retrieve all the data in the container requested. Index file is not used in this query execution. Query, which is given as input is validated, followed by container mapping and retrieval.

The algorithm which is used to execute a simple query is explained below

```
Procedure simple_query()
/* Simple query evaluation */
Begin
1. Con = tag id of the last node
2. Open con.cont.cc file and the dictionary file.
/*Data retrieval*/
5. Load the dictionary
6. While not EOF
7.      Read the code from the container
8.      If (code = = '>')
            Print the text array
```

9. Else if (code = ='<')
Decode and print the number which says the count of empty records
10. Else if (code < 256)
Store the ASCII value in text array
11. Else
Find the prefix and append character of the code and store it in the text array
End while
End

8.2.1 EXAMPLE

For a simple query like
//PLAY/ACT/SCENE/SPEECH/SPEAKER
The above query is obtained as input, after validation, tag id of the last node is obtained and the particular container is opened to decompress and display all the records in it.
Thus the simple query which is taken as input, undergoes three different types of validation namely path, node and level. After it is validated, the container to be retrieved is identified and the decompressed data is displayed.

8.3 AGGREGATE QUERY

Aggregate query consists of functions like count, max, min in it. Based on the function given processing is done and the output is displayed to the user. Aggregate queries can be present in a nested query either as dependent or independent part.

Tokenizer splits the given query into tokens from which the conditions are extracted and find out the function used. Meta table is loaded, if Index file is available for the particular query's, then output is obtained using the index file. If there is no index file, traverse the metadata file and count the number of lines spoken by each speaker.

Traverse the Metadata file and count the number of lines spoken by each speaker.

(i.e count the number of 21s after each 20)

The algorithm which is used to execute an aggregate query is explained

Procedure aggregate_query()
/* aggregate query evaluation */
Begin
1. Extract the aggregate function and the condition given in the query
2. Based on the target tag, open the container to be processed
3. If (file opened is an index file)
4. For each line
5. If (value stored for the key satisfies the condition given)
Print the key.
End for
6. Else
7. While(not EOF)
If the given condition is likely to be satisfied, store it as the intermediate result
End While
1. Print the final results

End

EXAMPLE

For an aggregate query like

//PLAY/ACT/SCENE/SPEECH[count(LINE)>5]/SPEAKER

the query is obtained as input and validated. Since LINE tag is given as the argument for count function, index file for the entire speaker container is opened. For the index file given in Fig.3.5.1 the output will be

PHILO

CLEOPATRA

This is because, when the total number of lines spoken by PHILO is counted we get 6 (1+1 = 2 , 15 +3 gives 4 lines i.e starting from 15th line he has spoken 3 more lines). Likewise for CLEOPATRA it is 7 (3+5 gives 6 lines, 11+0 represents one single line). But for the rest of the speakers the count becomes less than 5. Only PHILO and CLEOPATRA records are printed. Thus referring Meta data file or traversing the speaker container fully without indexing increases the execution time which is not done here.

Thus the aggregate query which is given as input gets tokenized to extract the condition. Then it undergoes three different types of validation namely path, node and level. After it is validated, the container to be retrieved is identified and the corresponding index file is opened and the desired data is displayed.

8.4 CONDITIONAL QUERY

Conditional query has a condition specified for a particular tag. The records that satisfy the given condition are located using the index and they are decompressed and displayed to the user.

ALGORITHM

The algorithm which is used to execute a conditional query is explained

```
Procedure conditional_query()
/* conditional query evaluation */
Begin
1. Extract the condition from the query.
2. Open the container representing the given condition's tag.
3. If (file opened is an index file)
   Find the index where the condition occurs and store it in an array.
4. Else
   Traverse the file to find where the given condition occurs and store it in the array.
5. Traverse the Meta data file and find out the tags whose data has to be retrieved for the given condition.
6. While ( there is a tag )
7.Open the container with the tag id.
8.For(i=0, i<index.length, i+=2)
9.If(index[i]%50 != 0)
      cont_no=index[i] / 50 +1
      record_no = index[i]%50
10.Else
      cont_no=index[i] / 50
      record_no = 50
```

Decompress the records starting from record_no (e.g. record 1) to index[i+1] records in the container cont_no.cont.cc(e.g. 21.cont.cc) and print.
End for
End while

End

EXAMPLE

For a conditional query like

//PLAY/ACT/SCENE/SPEECH[SPEAKER='PHILO']

the query is obtained as input and validated. Since the SPEAKER tag is given, for checking the condition, index file for the entire speaker container is opened. If we refer to the index file given in Fig.3.5.1, initially the numbers 1,1,15,3 as mentioned for PHILO are stored in an array.

E.g. arr[0]=1, arr[1]=1, arr[2]=15,arr[3]=3,arr[4]='\0'

Container to be opened is found from every even element and the number of lines to be printed in that container is identified from the successive odd element. Referring to procedure conditional query, since 1%50 != 0, container to be opened is 1/50 +1 i.e. 1st container. Starting record number is 1/50=1. Number of records to be printed from there is arr[1] = 1. Thus records 1 and 2 are decompressed and displayed. Similarly starting from 15th line, 3 more lines are printed. Then, the tags (e.g.STAGEDIR) under SPEECH node are identified from the Meta data file as the Meta data file tells the order of the tags as they are present in the original file. The indexes of the tags to be printed are stored in the array and the corresponding contents are decompressed and displayed.

8.5 COMPOUND QUERY

Compound query has more than one condition in it. There can be two or more tags with the condition specified. The records that satisfy the given conditions are located using the index and they are decompressed and displayed to the user.

ALGORITHM

The algorithm which is used to execute a compound query is as follow

Procedure compound_query()
/* compound query evaluation */
Begin
1. Extract the conditions given in the query.
2. Open the container representing the given condition's tag.
3. If (file opened is an index file)
Retrieve the index from the file based on the condition given and store it in an array.
4. Else
Traverse the file to find where the given condition occurs and store it in the array.
5. Traverse the Meta data file and find out the tags whose data has to be retrieved for the given condition.
6. While (there is a tag)
7.Open the container with the tag id.
8.For(i=0, i<index.length, i++)
9.If(index[i]%50 != 0)
cont_no=index[i] / 50 +1
record_no = index[i]%50

10.Else
cont_no=index[i] / 50
record_no = 50
Decompress the record record_no in the container cont_no.cont.cc and print.
End for
End while
End

EXAMPLE

For a compound query like

//PLAY/ACT/SCENE/SPEECH[SPEAKER>='MARK ANTONY'&&SPEAKER<'PHILO']

The query is obtained as input and validated. Since SPEAKER tag is given for checking the condition, index file for the entire speaker container is opened. If we refer to the index file given in Fig.2, initially the number of the first line spoken by MARK ANTONY is obtained and that is 9 here. Then, number of the starting line spoken by PHILO at the last instance is identified and that is 15 here. Now lines 9,10,11,12,13,14 are decompressed and printed. Then, the tags (e.g.STAGEDIR) under SPEECH node are identified from the Meta data file. The indexes of the tags to be printed are stored in the array and the contents are decompressed and displayed.

8.6 NESTED QUERY

Nested query has more than one query nested inside. There will be dependent and independent parts of the query. At first, the independent queries are extracted and executed and based on the value retrieved, dependent parts of the query are run. The records that satisfy the given conditions are located using the index and they are decompressed and displayed to the user.

ALGORITHM

The algorithm which is used to execute a nested query is explained below

Procedure nested_query()
/* nested query evaluation */
Begin
1. Extract the independent part of the query.
2. Execute the independent query and find the result.
3. Have this result as the condition value for the dependent query
4. Find the index of the records of interest.
5. Open the container with the desired records.
6.Traverse the records
7.For(i=0, i<index.length, i++)
8.If(index[i]%50 != 0)
cont_no=index[i] / 50 +1
record_no = index[i]%50
9.Else
cont_no=index[i] / 50
record_no = 50
Decompress the record record_no in the container cont_no.cont.cc and print.

End for
End

EXAMPLE

For a nested query like

//PLAY/ACT/SCENE/SPEECH/SPEAKER=[//PLAY/ACT/SCENE/SPEECH[LINE='The office and devotion of their view']/SPEAKER]/LINE the query is obtained as input and validated. The independent part of the query '//PLAY/ACT/SCENE/SPEECH[LINE='The office and devotion of their view']/SPEAKER' is separated and evaluated. LINE container is opened and the line number of the line 'The office and devotion of their view' is found out. Now, the index file is opened and the name of the speaker who has spoken that line and the index of the rest of the lines spoken by him are retrieved. Finally, LINE container is opened and the lines are decompressed and displayed to the user. This query can also be thought of as a correlation predicate query, because it compares a local context and an enclosing context. Correlation predicate queries in [7] are referred. Oracle takes time to execute nested queries because for each and every row in the outer table, it fetches each and every row in the inner table. Thus in order to reduce the execution time, a new approach [7] is proposed in which the inner query is executed first and then the outer query is executed with the result of the inner query.

9. PERFORMANCE ANALYSIS

9.1 SPECIFICATIONS

The open source datasets, Shakespeare and Lineitem, is used to test and evaluate our system. Test machine was an Intel Pentium core 2 duo @ 2.53 GHz speed, running Windows XP. RAM capacity is 2GB.

9.2 DATASETS

The following are the datasets used to evaluate our technique.

DBLP: DBLP stands for Digital Bibliography Library Project. Bibliographic information on major computer science journals and proceedings is given.

Shakespeare: All Shakespeares plays are converted into a single XML file and it contains long textual passages.

SwissProt: SWISS-PROT which is a protein sequence database strives to provide a high level of annotations (such as the description of the function of a protein, its domains structure, post-translational modifications, variants, etc.), a minimal level of redundancy and high level of integration with other databases.

Lineitem: The biggest file from TPC-H Relational Database Benchmark (Converted to XML by Zack Ives).

9.3 STORAGE SIZE COMPARISON

The compression ratio of IQCX and other queriable compressors are shown in Table 1.and the corresponding graph obtained is shown in Figure 8. It is proved that QUICX after indexing is efficient with highest compression ratio.

Table 1 Storage size comparison of IQCX with other queriable compressors.

STORAGE TECHNIQUE	SHAKESPEARE	SWISSPROT	DBLP	LINEITEM
XBZIP INDEX	21.83	7.87	14.13	-
XBZIP	17.46	4.66	9.69	-
RFX	15.79	8.77	7.638	-
QUICX(Before Indexing)	36.96	57.8	61.8	74.5
IQCX(After Indexing)	37.82	67.9	68.3	80

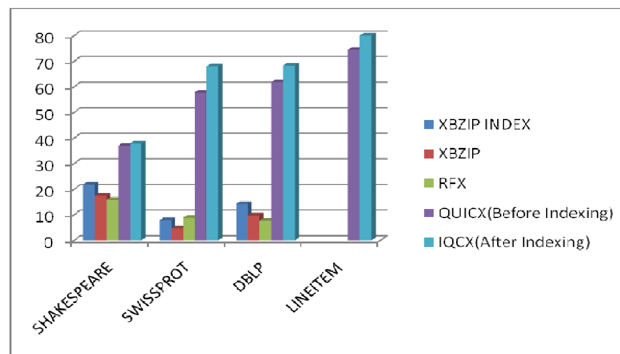


Figure 8. Storage size comparison of IQCX with other queriable compressors.

Thus it is shown that the compression ratio gets increased due to the proposed indexing technique.

9.4 Querying using the index

Simple queries which do not use the index are executed and the querying time for IQCX and other compressors are shown in Table 2. and they are compared as shown in Figure.9. Simple queries are taken from [13].

Shakespeare.XML

Q1. /PLAYS/PLAY/TITLE
Q2. /PLAYS/PLAY/ACT/SCENE/STAGEDIR

Swissprot.XML

Q3. /root/Entry/@id
Q4. /root/Entry/Ref/Comment

Lineitem.XML

Q5. /table/T/L_ORDERKEY

Q6. /table/T/L_COMMENT

Dblp.XML

Q7. /dblp/article/cdrom

Table 2 Simple query execution time comparison of IQCX with other compressors

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
XSAQCT	0.65	1.06	7.93	9.08	1.96	5.49	10.92
TREECHOP	1.56	1.44	17	16.63	5.06	6.22	15.49
IQCX	0.015	0.469	3.031	5.14	0.250	3.844	0.453

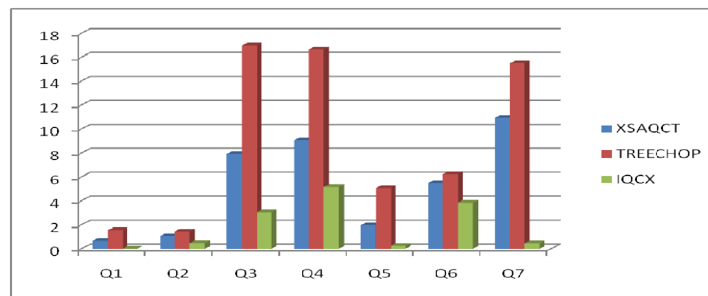


Figure. 9. Simple query execution time comparison of IQCX with other compressors

Aggregate query is executed using the index file generated.
Q8. //SPEECH[![STAGEDIR]]/SPEAKER/$C
Q9. //L_DISCOUNT/$U
The above queries are executed and the execution time is compared with XQzip as shown in Figure.9 and it is proved that our indexing technique greatly reduces the time required to process the query.

Q10. //PLAY/ACT/SCENE/SPEECH[count(LINE)>15]/SPEAKER

To execute this query, Meta data file need not be traversed. It is enough to traverse the index file generated and count the number of lines spoken by each speaker and if it is greater than 15, the speaker's name is sent to the output. The overall time taken for indexing is 6 seconds. The overall query execution time is found to be 0.187 seconds and this will be less compared to traversing Meta data file every time, performing calculation and retrieving speaker names and the querying times are shown in Table..3 and it is corresponding graph is shown in Figure. 10

Table 3 Querying time comparison of Aggregate Queries

	Q8	Q9	Q10
Xqzip+	0.005	0.032	-
IQCX	0.001	0.0063	0.187

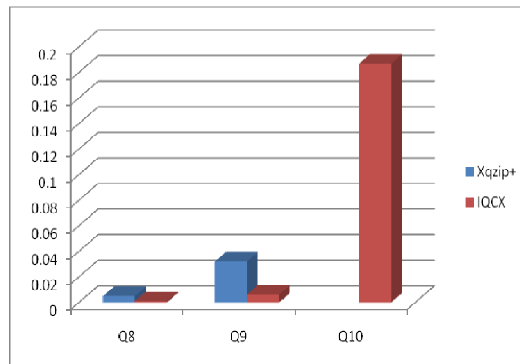


Figure 10 Querying time comparison of aggregate queries

Thus IQCX approach greatly reduces the execution time of aggregate queries compared to other compressors.

Conditional queries from [4] are taken and the execution times are compared.

Conditional query is executed using the index file generated.

Q11. //PLAY/ACT/SCENE/SPEECH[SPEAKER='PHILO']

Q12. /table/T[L_TAX='0.02']

Q13. /dblp/inproceedings[booktitle='SIGMOD Conference']

To execute this query, Meta table and the dictionary file are loaded initially. When the query is validated, it is split into tokens and the condition (SPEAKER='PHILO') is extracted. Index file is opened and the index of the lines spoken by the particular speaker is taken. Only those lines are decompressed and displayed from the lines container. The contents of the tags under speech node with speaker 'PHILO' are displayed. Time taken to execute this query is compared with other compressors as shown in Figure.11.

Table 4.Querying time comparison of Conditional Query

	Q11	Q12	Q13
XQzip-	0.038	0.044	0.345
XGRIND	1.620	2.890	26.108
IQCX	0.015	0.036	0.047

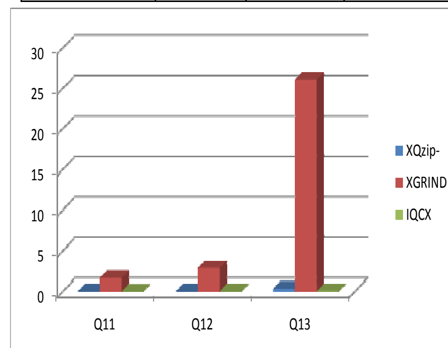


Figure.11 Querying time comparison of conditional query

Compound query which has more than one condition is executed using the index file generated.
Q14.//PLAY/ACT/SCENE/SPEECH[SPEAKER>='MARK ANTONY'&&SPEAKER<'PHILO']
Q15. /table/T[L_TAX>='0.02'&&L_TAX<='0.04']
Q16. /dblp/inproceedings[year>='1998'&&year<='2000']

To execute this query, Meta table and the dictionary file are loaded initially. When the query is validated, it is split into tokens and the condition (SPEAKER>='MARK ANTONY'&&SPEAKER<'PHILO') is extracted. Index file is opened and the starting line number for the speaker given (MARK ANTONY here)and the ending line number for the given speaker(PHILO here) are found out. Only those lines within the start and the end are decompressed and displayed from the lines container. The contents of the tags (e.g STAGEDIR) under speech node within these lines are displayed. Time taken to execute this query is compared with other compressors as shown in Figure 12.

Table 5. Querying time comparison of Compound Query

	Q14	Q15	Q16
XQzip-	0.039	0.075	9.541
XGRIND	2.312	3.210	50.344
IQCX	0.031	0.063	1.000

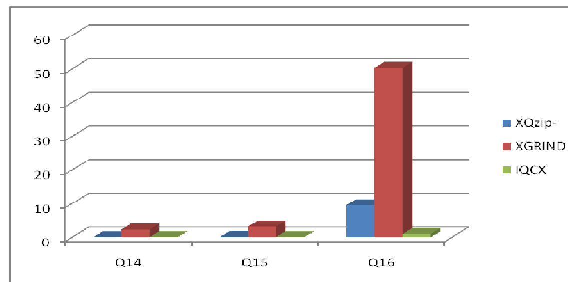


Figure. 12. Querying time comparison of Compound Quer y

Nested Query, which has a query nested in it is executed using the index file. The independent part of the query is executed first and the result is retrieved. Then it is used for executing the dependent part of the query.

Q17.//PLAY/ACT/SCENE/SPEECH/SPEAKER=[//PLAY/ACT/SCENE/SPEECH[LINE='The office and devotion of their view']/SPEAKER]/LINE
The query is got as input, validated and speaker who spoke the given line is found out. Finally the lines spoken by the particular speaker are decompressed and displayed.
Thus it is proved that the indexing technique proposed greatly reduces the time required to locate the records of interest which in turn reduces the execution time and increases the querying efficiency.

10. Conclusions and Future Work

This indexing technique greatly reduces the overall storage size required. The index file is later used for evaluating queries. It is also proved that the overall time required to index the file once and use

them for evaluating queries is lesser when compared to using the original files without indexing. The future work involves query optimization and evaluating queries using the index if required and prove that the overall execution time gets reduced if index file is used in locating records. It also involves runtime updation of the XML file and evaluating queries using intra and inter documents.

REFERENCES

[1] Arzucan O zgur, Taflan _I. Gundem (2006) "Efficient indexing technique for XML-based electronic product catalogs" , Electronic Commerce Research and Applications 5 , 66–77

[2]. Bao, Zhifeng; Lu, Jiaheng; Ling, Tok Wang; Chen, Bo (2010) " Towards an Effective XML Keyword Search ", Knowledge and Data Engineering, IEEE Transactions on Volume: 22 , Issue: 8.

[3]. B.M. Monjurul Alom, Frans Henskens, Michael Hannaford (2010) "DRXQP: A Dynamic Relational XML Query Processor" Seventh International Conference on Information Technology.

[4]. CHENG, J. AND NG, W. (2004) "XQzip: Querying compressed XML using structural indexing" Proceedings of the International Conference on Extending Database Technologies. Heraklion, Greece, 219–236.

[5]. Gregory Leighton, Tomasz Müldner, James Diamond, (2005) "TREECHOP: A Tree-based Query-able Compressor For XML" , In Proceedings of the Ninth Canadian Workshop on Information Theory (CWIT 2005), pages 115-118.

[6]. Liang Jin,Nick Koudas,Chen Li,Anthony K.H.Tung (2007) " Indexing Mixed types for appropriate retrieval" Proceedings of the 31st VLDB conference, Norway.

[7]. Matthias Brantner, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, "Kappa-Join: Efficient Execution of Existential Quantification in XML Query Languages".

[8]. Ming Li,Murali Mani, Elke A. Rundensteiner, (2008) " Efficiently loading and processing XML streams", Proceedings of the 2008 international symposium on Database engineering & applications.

[9]. Nils Grimsmo,(2008) "Faster Path indexes for search in XML data", Proc.19 Astralasian Database Conference.

[10]. Paolo Feragina, Giovanni manzini (2007) "Indexing Compressed text", Journal of the ACM,Vol.52,No.4,July 2007.

[11]. P. Ferragina, F. Luccio, G. Manzini, S. Muthukrishnan, (2006) " Compressing and Searching XML Data Via Two Zips", International World Wide Web Conference Committee, WWW 2006.

[12]. Radha Senthikumar, A.Kannan" Query and Update support for Indexed and Compressed XML (QUICX)". In proceedings of 1st International conference on computer science, Engineering & Application (ICCSEA 2011), UAE, WIMON/CoNeCo, CCIS 162, Springer-Verlag Berlin Heidelberg 2011, pp 414-128.

[13]. Radha Senthikumar , S. Priyaa Varshinee, S. Manipriya, M. Gowrishankar, A. Kannan, (2008) "Query Optimization of RFX Compact Storage using Strategy List", ADCOM 2008.

[14]. Tomasz, Müldner, Christopher Fry, Jan Krzysztof Miziołek and Scott Durno (2009) "XSAQCT: XMLQueryable Compressor." Presented at Balisage: ThE Markup Conference 2009, Montréal, Canada, August 11 - 14, 2009. In Proceedings of Balisage: The Markup Conference 2009. Balisage Series on Markup Technologies, vol. 3.

[15]. Zafari. H, Hasani. K, Shiri, M.E., (2010) "XLight, An Efficient Relational Schema to Store and Query XML Data ", Data Storage and Data Engineering (DSDE).