World Journal of Innovation and Modern Technology E-ISSN 2756-5491 P-ISSN 2682-5910
Vol 8. No. 4 2024 www.iiardjournals.org Online Version

A Constraint Identification Method for Predicate Node Identification in Clustered Xml Documents

[1]B.A. Bodinga, [1]A. Roko, [1]A.B. Muhammad, [2]I. Saidu
[1]Department of Computer Science, Usmanu Danfodiyo University, Sokoto-Nigeria
[2]Department of ICT, Usmanu Danfodiyo University, Sokoto-Nigeria
*Corresponding Author: bello.bodinga@ududok.edu.ng
DOI: 10.56201/wjimt.v8.no4.2024.pg137.151

Abstract
A large number of documents are now represented and stored using an XML document structure on the web. These documents may emanate from the same source (Homogeneous) or different sources (Heterogeneous). This make it challenging as how these documents can be managed and retrieved. The existing systems returns irrelevant predicates. The predicate node identification method employed on the search systems use only a simple constraint. To improve the effectiveness of XML retrieval, an effective constraints Identification Algorithm (E_CIA) is developed to identify relevant predicates. The E_CIA uses Constraints Operator Generator (COG) to identify constraints to be imposed to generate most relevant predicate node to improve the effectiveness of the retrieval process. Experiments have been conducted to evaluate the performance of the proposed E_CIA. The experimental results have shown that the proposed E_CIA outperforms StruX and StruXPlus in terms of precision.

Keywords: *XML retrieval, Constraint Identification, Predicate node, Constraint Operator Generator*

1. Introduction

To retrieved data stored in XML document, query languages are developed. Using query language, a user can compose his query to retrieve data from XML documents. Such queries are called structured queries. Formal languages or structured query languages such as XPath or XQuery are used to query the collection of XML documents to retrieve relevant parts of the XML document ([8], [10]). XQuery is a functional language that is used to retrieve information stored in XML format. It is similar to SQL in relational database. XQuery has an expression called a FLWOR (for, let, where, order by, and return) expression, used to query XML documents and return different parts of the document. For example, Figure 1 below is XQuery's FLWOR expression that returns 'movies' where 'writer' contains 'Bralver Bob'. This is similar to an SQL select statement where movies is a **table** (called entity in XML retrieval) and the where clause is writer = 'Bralver, Bob' is a filter condition, which we called constraint condition, in this research.

```
for $c in collection('imdb.xml')//movies
where $c/ writer = 'Bralver, Bob'
return $c
```

Figure 1: A XQuery Based on single condition

In the preceding query, the 'for' clause binds the variable $c to each movie element and the 'where' clause filters out bindings of the variable $c for which the writer element contain the value 'Bralver, Bob'. The return clause produces a result for each binding of $c.

In XML retrieval community, the 'movies' node is called an entity node and 'writer' node a predicate node [3]. Predicate node is like a field of table in relational database and 'Bralver, Bob' is one of its content. Because an entity node is more informative than the predicate node, entity nodes are often returned as query results in XML retrieval [1]. Since, several entity nodes can be returned as answer to a query even including irrelevant nodes, the predicate nodes and their contents are then used to filter out the irrelevant entity nodes. For example, writer = 'Bralver, Bob' is a simple constraint condition which is used to filter out irrelevant nodes.

However, the constraint condition can be compound condition connected with AND/OR constraint operators [4]. For example, the query in Figure 1 can be re-written as follows:

(a)

```
for $c in collection('imdb.xml')//movies
where $c/writer = 'Bralver, Bob' AND $c/director = ' Pintoff, Ernest'
return $c
```

(b)

```
for $c in collection('imdb.xml')//movies
where $c/writer = 'Bralver, Bob' OR $c/director = ' Pintoff, Ernest'
return $c
```

Figure 2: XQuery based on compound conditions

The query in Figure 2 (a) returns 'movie' node based on writer and director. All movie nodes that satisfy the filter conditions are returned and the one in Figure 2 (b) returns movie node based on writer or director.

As noted by [4], filtering based on two constraints could return more relevant results. These make XQuery queries to be effective. However, for most users formulating XQuery queries prove to be difficult even for experienced users [7]. Because it requires the user to have formal knowledge of the query language and knowledge of the underlying XML document schema. Further, determining which logical operator (AND/OR) to be used to form compound constraints prove to be challenging [4]. This prompted the development of system that automatically generate queries developed ([6], [9], [11]). For example, the systems allow a user to enter a keyword query and it automatically converts the keyword query to XQuery query or another structured query. These systems are called Query Structuring System (QSS) and their process of conversion consists

of three steps: how to represent user queries, identifying entity and predicate nodes to improve keyword search. Several predicate identification algorithms are developed ([6], [9]). Existing Predicate Node Identification Methods utilizes only a single constraint condition which make them return many irrelevant predicate nodes.

This paper presents our constraint identification method for an effective Predicate Node Identification. The method consists of Constraint Operator Generator and Effective Constraint Identification Algorithm. The rest of this paper is organized as follow: section 2 presents related works, section 3 presents preliminaries, section 4 presents the proposed E_CIA, sections 5 and 6 presents Experimental results and lastly the paper concludes in section 7.

2. Related works

In order to accommodate the notion of more relevant elements and likely relevant elements caused by keyword query and the need to develop systems that are both efficient and effective prompted the development of Query structuring systems. In this section, two leading works on Predicate Node Identification which were benchmarked in this paper are presented by highlighting the operation, strength and weaknesses as follows:

StruX [6] does not dictate the way a user pose his/her queries. Given a keyword query, StruX generates a sequence of segments based on the keywords in the query, where a segment consists of one or more query keywords. Pairs of segments are then combined; each segment combination is labelled with elements from the XML data. This approach is impractical to be used for an environment where DTD is not part of the XML document because entity node in the target XML documents are computed based on the heuristics applied on the DTD. Another drawback to StruX is that the segmentation strategy employed does not consider the fact that a group of consecutive keywords represent a single named entity and therefore need not to be splitted. This idea was backed by [5], where a new approach called in-doubt-without approach was developed with the basic idea that, in cases of doubt, it is often better to (partially) leave queries without any segmentation. In addition, its ranking function assigns score to individual generated structured queries based on statistics of the keywords in the XML element node that form the query only and this make the system return lower quality result in terms of query relevance.

To address the drawbacks in StruX, the StruXPlus [9] was proposed. First, an entity-based query segmentation method (EBQS) that return correct query interpretation is introduced. Secondly, a segment term proximity scorer (STPS) is proposed to resolve keyword query ambiguity. Then an effective predicate Identification algorithm to infer relevant predicates is developed. However, the formula used to compute relevant predicates fails to be conscious of the semantic relatedness between terms. In addition, a simple constraint is employed which make it return many irrelevant predicates.

3. Preliminaries

This section presents building blocks for the proposed method. The section first introduces query segments and predicates in section 3.1, then explores the information need from a keyword query in section 3.2.

3.1 Segments and Predicates

Query Segmentation is one of the critical components for understanding users' search intention in Information Retrieval process [8]. This involves grouping tokens in the keyword query into meaningful phrases (segments) which help tasks such as search relevance and keyword query understanding.

XML keyword Search systems retrieves high quality and most relevant results when they are able to identify the important phrases in the keyword query which need to be kept together for quality results. One way to achieve this is for the user to be explicit about the phrases by adding quotes around the segments of the keywords query to indicate phrases. But this is not the pattern in the real-world search logs. Users expect the keyword search system to infer and understand these phrases. This ends up lowering the precision in most cases where the phrase as a whole is important to be kept together during retrieval process in IMDB dataset like movie name, song title, brands etc. For example, Consider a user's query q= '*XML Retrieval Mounia Lalmas'*. The user is looking for mainly '*XML Retrieval* document authored by *Mounia Lalmas'*. The underlying keyword search system needs to know that the query is for '*XML Retrieval'* and specifically written by '*Mounia Lalmas'* as an additional feature of the document. The search experience is different if a user searches with quotes around the segments -*"XML" "Retrieval" "Mounia Lalmas"* compared to the unquoted query. If the query is treated as a bag of words, the results might end up being less precise. In the keyword query '*XML Retrieval Mounia Lalmas'*, the results are far from being accurate if we show the user items which match *'XML Retrieval Mounia Lalmas'* with *'XML Mounia Lalmas Retrieval'*. Order plays a vital role in query segmentation which is lost in a bag of words model. A keyword query $q = \{k_1, k, \dots, k\}$ can be segmented to form a segment list with $S = \{s_1, s_2, \dots, s_k\}$ and each segment $s_i = \{t_1, t_2, \dots, t_n\}$ where t_i is a query term or a sequence of query terms that describe an entity. If $A = \{a_1, a_2, \dots, a_m\}$ is the schema, with a_i as an attribute of the XML document in question, a segment-attribute combination can be $a_1 = s_1$, $a_1 = s_3$, $a_3 = s_1$, ... Each of this combination is called a *predicate* according to some guidelines adopted from [1]. Entity nodes are used as the user search intention: *target* node and *predicate* node. *Predicate* node and *entity* node are then use to generate a set of structured queries [1]. The predicates are nodes that restrict the type of *entity* (*search-for*) nodes to be returned, also called *search-via* nodes.

Approaches in StruX [6] and StruXPlus [9] treated keyword query just as a set of keywords. However, in this research, XSeek [7] classification of keyword queries is adopted by grouping keywords in a query into two categories: (i) if a keyword *k* matches the value of a node or it matches the label of a node that has got a descendant matching another keyword, then *k* species a

predicate, corresponding to the WHERE clause in XQuery; (*ii*) otherwise, it is treated as a return node, corresponding to the RETURN clause in XQuery.
For example, consider a query *Q*= {Monsieur Vincent Cloche Maurice} on IMDB dataset which comprises of all predicates and each keyword is a predicate. We observe that there are relations between keywords in the query and usually the keywords can be combined together to form a predicate or return node. The query *Q* can be expressed as two predicates (:Cloche Maurice and title:\ Monsieur Vincent) and a return node (movie).

3.2 Exploring the Information Needs in XML Queries

In this section, we explore information need from a keyword query submitted by the user. We consider a user query as a target [11] which represent the piece of information the user is seeking for. As noted by [4], the target consists of content needs and the concept needs. The content needs (*CTN*) indicates the actual text the user is seeking for. For example, in the Xquery query in Figure 1, the content need is movies indicating that the user will like to see information related to movies written by 'Bralver, Bob' Which is a predicate node. The concept needs (CPN) includes the constraint keywords (CK) used to filter the relevant content needs (CTN) to narrow the result to be returned to the user and the constraint operator (COp) as shown in Figure 3. A compound constraint group element has one or more constraint elements (to indicate where disjunctive or otherwise) which consist of concept element, C, and a keyword element, K. The concept of constraint is not mandatory. In the case when the concept is not specified, its value is null. A compound constraint group element has a logical operator as its attribute. The logical operator attribute can either have 'OR' or 'AND' as its value, to indicate where its constraint elements are disjunctive or conjunctive.
The main idea behind this proposed method is to separate the CTN from CPN during query representation process. Our contribution in this work is introduction of logical constraint concept to filter out irrelevant nodes thereby improving precision.

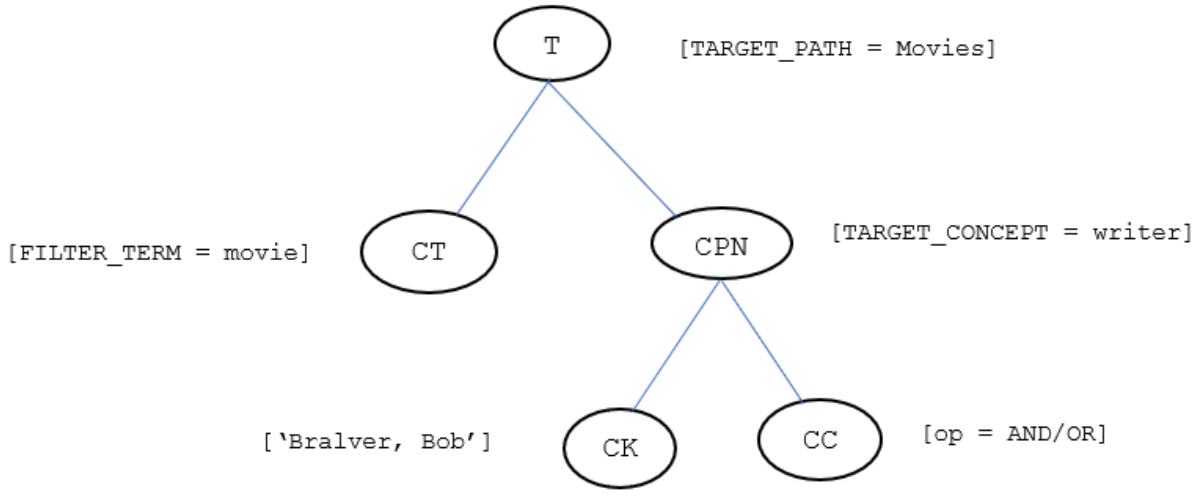

Figure 3: An Explored XML Information Need derived from XQuery in Figure 1

Different from StruXPlus, the proposed solution provides a constraint generator method to augment the Effective Predicate Identification. For example, in the XQuery query reformulated in Figure 2(a), logical operator **AND** is used to create a very narrow search on movies written by 'Braiver, Bob' and directed by 'Pintoff, Ernest'. User search intention consists of two types of structural constraints [9]: where to look (i.e. predicates) and what to return (i.e. return elements or entity nodes). Given a user query q consisting of n keywords, where $q = \{k_1, k_2, \dots, k_n\}$, for example a query to find movies written by 'Bralver, Bob' presented in the XQuery in Figure 1. The user information need is explored as presented in Figure 3 above.

3.3 Index Construction

This section describes the index construction. To construct this index, MariaDB database is created with four tables to store the index. This is what distinguishes our index with [9]. The database is shown in Figure 4 and the primary keys of the tables are shown underlined. The MariaDB database is compatible with popular MySQL database by adding support to more storage engines and fixing storage engines limitation.

Clust (clusID, clustName, noOfDocs)
Document (*docId, docName*)
nodeTb (*nid, epath, nType, noOfChilds*)
node_index (*docId, nid, dewId, freq, pos*)

Figure 4: The Index structure

The process of indexing is done as follows:
For each cluster in clusters, for each document *D* in cluster, the indexer first creates and returns a tree-like representation of the document. Each element in the XML document is represented as a node in the tree. While creating the tree, the indexer also allocates *deweyId* to each node in the document. Dewey Id is a sequence of digits that uniquely identify an XML node. The tree is traversed in pre-order and the following information are collected for each document *D* and for each node *n* in *D* visited:

i. Store the document ID (*docId*), cluster ID (*clustId*) and document *name* in the *document* table in Figure 4.
ii. Store the prefix path (*prefixPath)* of *n* as its node type in the *nodeTb* table in Figure 4. Each row in the table consists of the node's name (*epath*), nodes' number of children (*noOfchilds*), and node Id (*nId*), which is used to identify a node.
iii. In case *n* is a leaf node, for every term (*trm)* in *n*, a record is created in the *node index* table in Figure 4. This table is similar to a posting list. Each row in the table stores a detail of a node with respect to a particular *trm* in the node. A row is made up of the *nId*, the *trm*, the frequency (*freq*) of *trm* in the node, position (*pos)* of the term in the node, the NEC of the term *trm* (*category)*. Also, in the row, are the nodes' *dewId* and the *docId*. The *docId* is used to identify in which document the XML node belongs.

4. Proposed ECIA

In this section, ECIA that is an improvement on the EPIA [9] is presented. The section is divided into two: firstly, the shortcomings of the EPIA [9] and its predecessor the PIA [6] are presented in section 4.1. then the solution to the shortcomings are presented in section 4.2

4.1 Limitations of PIA and EPIA

Given a user keyword query, StruX [6] split the user query keywords into sequence of segments, where each segment consists of a query keyword or a sequence of keywords. In this case, the system ignores the fact that a query keyword can appear in different parts of an XML document having different semantics. It considers a query just as a sequence of keywords not as a sequence of semantically related terms. This problem causes the existing systems to return irrelevant predicate nodes from XML data. As a result, StruXplus [9] developed entity-based query segmentation (EBQS) method which first interprets a user query as a list of keywords and/or named entities to resolve ambiguity. Then, segment terms proximity scorer (STPS) that assigns relevance scores to XML fragments that contains query keywords is proposed. Fragments containing the keywords as interpreted by EBQS are assigned higher scores. Finally, an effective predicate identification algorithm (EPIA) which uses EBQS and STPS to return relevant predicates is introduced. The system performance is low because it uses simple constraint condition to filter out entity nodes. This causes the system to return large number of irrelevant entity nodes.

4.2 Proposed solutions to EPIA problems

To enhance the performance of the system, we proposed an effective constraint identification method. The method consists of two algorithms: First, the XML documents collection need to be clustered to group similar documents in to clusters. The WEClusterX method developed in the previous chapter is employed. Using the clusters returned by WEClusterX method, we compute AND/OR operator and use them to compose query with complex constraints. Complex queries are queries whose conditions are connected with AND/OR. This algorithm, which we called constraint_operator_generator (COG) aimed to augment the predicate identification method with correct logical constraint operator(s) to help in query constraint formulation is presented. The main motive behind the introduction of the COG is to group the constraints that belong to the same predicate node. Secondly, an Effective Constraints Identification Algorithm (ECIA) is developed. ECIA accepts the list of segments (*segments*), clusters (as returned by WEClusterX method in the previous chapter), constraint operator (as computed by COG) as input and produces list of predicates (*ListPred*) as output.

Constraint Operator Generator Algorithm

In this subsection, a constraint operator generator algorithm is developed. The pseudocode is shown in algorithm 1

```
Algorithm 1:  Constraint Operator Generator (COG)
Input:        clusters, ListPred
Output:     constraint operator
1.for cluster in WEClusterX://The clustering solution proposed in [2]
2.       inside = [ ]
3.       for predicate in ListPred:
4.              if predicate in cluster
5.                    inside. append (True)
6.       end for
7.      if all (inside):
8.         return "AND"
8.         break
10.       return "OR"
```

Line 1 uses the clustering method proposed in [2] to cluster the dataset. On line 3, it calls the predications (segments, dataset) method which using query segments and dataset extract relevant predicates from the dataset that are to be returned to the user. Lines 3-9 check for every predicate in the list of predicates and if it finds the predicate in the cluster update list to True. Line 7 applies the python function *all* and if *all* are, then return "AND". Line 9 break and 10 return the otherwise "OR".

Effective Constraint Identification Algorithm

This section describes the proposed Constraint Identification Algorithm which helps filter out irrelevant predicate nodes. This algorithm takes the XML document (dataset), user query (qry), list of predicates (ListPred) and segments as input and produce logical constraints as output. This method is based on the intuition that a relevant predicate is a node that contains at least one segment in close proximity [9].
Intuitively, in a relevant document, query terms appear relatively close to each other and not in completely unrelated parts of the document [8]. Adopting this intuition to XML documents, in this study, the proximity of query terms is calculated as follows:

$$stp(x,d) = \sqrt{\sum_{i=1}^{n}(x_i - ave)^2/n} \qquad 1$$

where *x* is a list of query term positions within an XML element *e*, x_i represents a term position in *e*, *ave* is the mean of the query term positions, and n is number of query terms used.

stp(x, d) computes how scattered the terms in a query are within an XML leaf element. A small value of *stp(x, d)* means the query keywords are clustered together, and a large value indicates the query keywords are widely separated. The introduced *stp(x, d)* has the following properties:

- Is always non-negative. Also, it is defined regardless of whether $x_1 - x_2 > 0$ or $x_1 - x_2 < 0$. This property is desirable since negative scores are not used.
- Becomes higher as $x_1 - x_2$ increases and vice versa; i.e., it rewards term proximity (since the score is inversely proportional to this quantity).
- Becomes higher if $x_1 and\ x_2$ are far apart i.e. if $x_1 - x_2 < 0$ and vice versa. This property indicates that $std(x, d)$ rewards correct term ordering.

In this method, equation (2) is adopted from [9] to compute relevant predicates for a given segment.

$$score(n, s) = p_s^n \log_e \left(1 + \sum_{k \in s} f_{n,k}\right) \quad (2)$$

Where *n* is a predicate node, s is a segment and *k* is a keyword contained in s. The P_s^n is a segment proximity scorer of segment s in node n. The first multiplier P_s^n in equation 2 computes XML nodes containing at least one of the keywords in s. The second multiplier (SPS) rewards nodes having the segment keywords in close in close proximity and in the order, they appear in segment s. As the terms are in close proximity with each other and in the same order they appear in s. The p_s^n is the segment proximity score of segment *s* in node *n* and is computed using equation 3. From Equation (2), if the value of $score(n, s)$ is non-negative, then *n* and *s* are relevant.

$$p_s^n = \frac{1}{\sqrt{\sum_{t=1}^{n}(pos\_i - ave)^2}}\Big/_t \quad (3)$$

where *t* is the number of keywords in segment *s*, *Pos_i* is the position of keyword *i* in s, and *ave* is the mean of the position of the first and the last keywords in s, which is computed using Equation (4) with β = 1.01, which is little bit greater than 1 to ensure that query term t_i is clustered around t_{i+1}in segment *s*. p_s^n score determines how the terms in the segment *s* are scattered within the node. The closer the terms within a node, the higher is the score. By including the notation of term proximity, the node having the segment keywords closer to each other gets bigger.

$$\text{ave} = \frac{(\beta^2+1) \cdot P_{t_{i,e}} \cdot P_{t_{i+1,e}}}{\beta^2 \cdot P_{t_{i,e}} + P_{t_{i+1,e}}} \quad (4)$$

where $P_{t_{i,e}}$ represents position of term t_i in *e* while $P_{t_{i+1,e}}$ represents position of term t_{i+1} in *e*.
The following example demonstrates how it works. Consider a query 'movie 2004 Bralver Bob ' issued on IMDB dataset, and the query is intended to search for a *movie* whose *writer '* Bralver 'and year '*2004'* and one of the actors is *'Bob'*. Using Equation (2), E_CIA returns the following three predicates each with its scores.

```
/movies/movie/year = '2004'        1.01654236

/movies/movie/writer = 'Bralver'    2.321343

/movies/movie/actors/actor = 'Bob'     1.1453901
```

Figure 5: Predicates and scores

The second predicates got the highest score because there are more */movies/movie/writer* nodes in the IMDB that contain the segment keywords than the */movies/movie/actor* nodes. But the user search intention does not include */movies/movie/actor* node.

The pseudocode of the E_CIA is shown in Algorithm 2. The algorithm works as follows: Line 1 retrieves all the nodes in the XML document collection using getAllNodeTYPE () and for each node, its relevant segments are computed from the given list of segments on lines 2-10. Line 2 iterates through all nodes in allnodesType and a list of relevant segments is computed for each segment s. The method sets *Rsegments* to empty since no any relevant segment yet found. In line 4, the method considers a segment s and compute a score for that segment in respect to n in line 5. Line 6 checks if the score of the segment s is greater than zero and update *Rsegments* in line 7, otherwise it iterates back to line 4 Line 8 selects the best segments while lines 9-10 first check if the list of segments is non-empty then it generates a list of related *node/segment* pairs called predicates. This process continues until all the nodes are considered. Finally, line 11 selects the best predicates and store the result in list of predicates *(ListPred)*. line 12 call the generate_AND_OR to generate all the logical constraint operators and returns the list of best predicates as the answer.

```
Algorithm 2:  Effective Constraints Identification
Algorithm (E_CIA)

Input:    segments List – segment, dataset
Output:  constraints

1.  allnodesType = getAllNodeTYPE ();
2.  for each NodeType  n  in allnodesType do
3.   Rsegments = null; //relevant segments for node n
4.   for each segment s in segments List do
5.        score = log(1 + Σ_{k∈s} f_{n,k}) P_s^n  // adopted from [9]
6.        if (score > 0) then
7.          Rsegments.add (s, score)
8.  if (Rsegments != null) then
9.        segment = selectBestSegment (Rsegments)
10.       predList.add (n, segment, score)
11  ListPred = genbestPredicates (ListPred)
12  clusters = WEClusterX (dataset)
13. ans = generate_AND_OR (ListPred, clusters)
13. if constraint = ""
14. for i in PredList
15.     if i < len(ListPred)
16.        constraint + ListPred [i] + " " + ans
17.     else
18.        constraint = constraint + ListPred [i]
19. return constraint
```

5. Experiments and Evaluation

To evaluate the performance of the proposed COG and E_CIA algorithms, we designed and performed a comprehensive set of experiments on real datasets. Our approach is compared against the state-of-art proposals of StruX [5] and StruXPlus [8]. The algorithms were used in StruXPlus and a new version called StruX++ is presented. For the conduct of experiments, Python programming language is used for the implementation on windows 10 professional 64-bit operating system. Niagara and DBLP datasets were used to evaluate the method. Experiments were conducted to investigate performance of the proposed *StruX++ using* an effective constraint identification method. We compare our proposed method against StruX [5] and StruXPlus [8] in order to test the effectiveness of our constraint operator generator (COG) algorithm along with the utilization of the previously proposed E_CIA using queries and relevant judgements. The queries and relevant judgments (which we called returned node) are obtained as follows: Twenty-five (25) keyword queries are randomly selected as done in [8] and a survey involving forty-five (45) students (both undergraduate and postgraduate) is conducted. In the survey, the students were asked to write the target XML predicate nodes that would be returned by each query. The survey

result is summarized and the 10 queries where more than 70% of the participants agree on the same target node are selected. These queries are indicated on column 2 and their corresponding predicate nodes (i.e. relevant judgments) on column 3 of both table 1 and 2 respectively. We chose StruX and StruXPlus because they are the most recently proposed predicate identification methods in XML retrieval. Niagara and DBLP datasets are used for the evaluation of *StruX++*. Performance evaluation results are presented using tables and in charts. Evaluation result on Niagara dataset is indicated in table 1 and in figure 6 while on DBLP dataset, it is indicated in table 2.and figure 7 respectively.

Table 1: Query evaluation result on Niagara dataset

Query Id	Query	Returned node	StruX	StruXPlus	StruX++
QN1	StoneBreaker Eugenia	Club	Club	Club	Club
QN2	Addison Wesley problem solving with C++	Publisher	Title	Author	Publisher
QN3	Nosperatu, eine Symphonie des Grauens	Movie	Movie	Movie	Movie
QN4	Casablanca	Movie	Movie	Movie	Movie
QN5	Goet Graefe Rule Based query optimization in extensible database systems	Title	Author	Title	Title
QN6	City Lights	Movie	Movie	Movie	Movie
QN7	Crafting a compiler with C	Book	Book	Book	Book
QN8	Nearman Ronald	ClubMember	ClubMember	ClubMember	ClubMember
QN9	1234 Yellowstone drive BeaVerton	Address	Address	Address	Address
QN10	Raghu Ramakri Ishnan	Person	Person	Person	Person

Table 2: Query evaluation result on DBLP dataset

Query Id	Query	Returned nodes	StruX	StruXPlus	StruX++
QD1	53 ACM Computing Arbitrary functions of encrypted data	Issue	Issue	Issue	Issue
QD2	PIM Meets web 2.0	Conference	Conference	Conference	Conference
QD3	51 11 Jim Gray, astronomer	Issue	Issue	Issue	Issue

QD4	Alexander Szalay	Author	Author	Author	Author
QD5	10 Peter Wood Query Relaxation in RDF	Issue	Issue	Issue	Issue
QD6	NAA 2008, Lozenetz	Conference	Conference	Conference	Conference
QD7	Chen Li 007	Author	Author	Author	Author
QD8	Performance evaluation in databases	Article	Journal	Article	Article
QD9	Numerical Quandrature	Article	Articles	Article	Article
QD10	Roots of the western tradition	Article	Author	Author	Article

6. Results Analysis and Discussions

All the queries are evaluated and results are shown in Tables 1 and 2 respectively. The first columns represent the query Id; the second columns represent the search value; the third columns represent the expected returned nodes. The fourth column represent the set of nodes returned by the StruX [5] the fifth columns represent the answers (i.e. nodes) returned by the StruXPlus [8]; while the last columns represent the set of nodes returned by the proposed system.

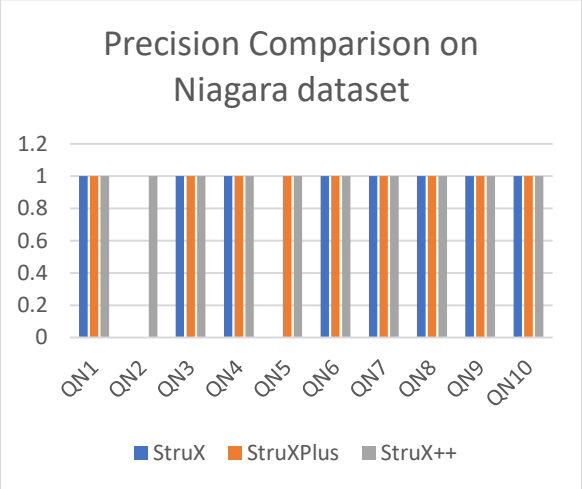

Figure 6: Precision comparison on Niagara Dataset

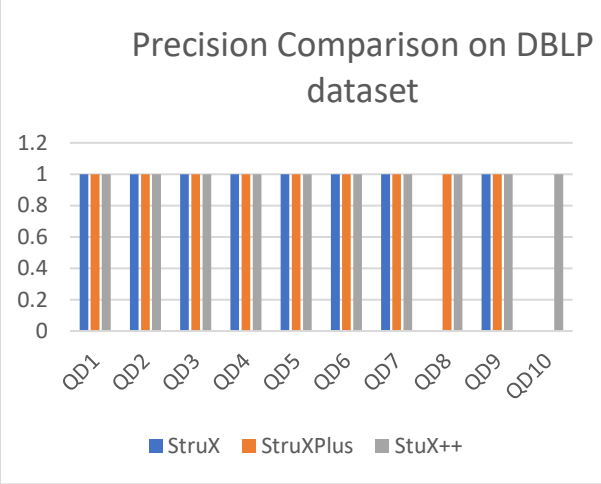

Figure 7: Precision comparison on DBLP Dataset

Figures 6 and 7 illustrates the precision comparison of the proposed method compared with StruX and StruXPlus using Niagara and DBLP datasets, respectively. The Figures shows that the *StruX++* achieves better search performance than both the *StruX* and S*truXPlus*. Figure 5.6 shows that *StruX++* is able to infer about 100% of the true return nodes while the StruX only 80% and the StruXPlus returns 90% on Niagara dataset. While Figure 5.7 demonstrates that the proposed

method infers about 100% of the return nodes and the StruX about 80%, and the StruXPlus obtains 90% on DBLP dataset.

The *StruX++ uses* the WEClusterX method proposed in Bodinga *et al.*, 2024 and the E_CIA and COG algorithms which are proposed to provide correct logical constraints resolve polysemous ambiguity. The algorithms are used in *StruXPlus to* produce *StruX++*.

The *StruX++* system performs better than the StruX and the StruXPlus systems because the system returns nodes that contain enough relevant information using its trained set to obtained semantic relatedness between nodes. This can be attributed to the fact that the proposed *StruX++* is a word embeddings based method and it acquires enough semantic information of the terms used to query the datasets.

7. Conclusion and future work

This paper presented the proposed *StruX++* using an enhanced method to predicate node identification for XML retrieval in heterogeneous environment. Experiments has been conducted to evaluate the performance of the proposed system. The results have demonstrated that the proposed StruX++ system outperforms the compared StruX and StruXPlus in terms the quality of desired returned nodes. In the future, we intend to develop a dynamic labelling scheme that allows XML Documents update so that the system can be used in dynamic environment.

REFERENCES

[1] Bao, Z., Lu, J., Ling, T. W., & Chen, B. (2010). Towards an Effective XML Keyword Search. *IEE Transactions On Knowledge And Data Engineering*, Vol. 22(8), pp.1077–1092.

[2] Bodinga, A. B., Roko, A., Muhammad, A.B., and Saidu, I. (2024). An Effective XML Documents Clustering Method Using Word Embeddings for Heterogeneous Collections. International Journal of Computer Science and Mathematical Theory. DOI: 10.56201/ijcsmt.v10.no2.2024.pg120.140

[3] Fuhr, N., Lalmas, M., & Kazai, G. (2002). INEX: Initiative for the Evaluation of XML retrieval. In *University of Dortmund*. article.

[4] Gan, K.H. and Phang, K.K. (2017). A Sematic-Syntax Model for XML Query Construction. *International Journal of Web Information Systems*. Vol. 13(2). doi: 10.1108/IJWIS-06-2016-0034.

[5] Hagen, M. Potthast, M., Stein, B. & Brautigam, C. (2012). The power of Naïve query segmentation. *In the Proceedings of the SIGIR'10 Conference*, Geneva, Switzerland. Pp. 1-2.

[6] Hummel, F., da Silva, A.S., Moro, M.M., & Laender, A.H.F. (2011). Automatically Generating Structured Queries for XML Keyword Search. *In S.Geva et.al., (Eds.): INEX 2010*, lncs 6932,194-205. Springer-Verlag Berling Heldeiberg.

[7] Liu, Z., and Chen, Y. (2007). Identifying meaningful return information for XML keyword search. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data. SIGMOD '07*. Pp. 320-329. New York, New York, USA: ACM Press.

http://doi.org/10.1145/1247480.1247518.

[8] Roko, A., Doraisamy, S., Jantan, A, H. and Azman, A. (2015). Effective Keyword Query Structuring using NER for XML Retrieval. *International Journal of Web Information Systems*, vol. 11 (1), pp. 33-53.

[9] Roko, A., Doraisamy, S., and Nakone, B. (2018). Effective Predicate Identification Algorithm for XML Retrieval. *In proceedings of the Fourth International Conference on Information Retrieval and Knowledge Management* (CAMP), Kota Kinabalu,Malaysia, 2018, pp. 1-5, doi: 10.1109/INFRKM.2018.8464696.

[10] Woodley, A., and Geva, S. (2006). Nlpx at inex 2006. *In N. Fuhr, M. Lalmas, & A. Trotman (Eds.),* Inex, 4518, 302-311. Springer-Verlag Berling Heldeiberg.

[11] Petkova D., Croft W.B., and Diao Y. (2009). Refining Keyword Queries for XML Retrieval by Combining Content and Structure. *In: Boughanem M., Berrut C., Mothe J., Soule-Dupuy C. (eds) Advances in Information Retrieval.* ECIR 2009, 5478. Springer, Berlin, Heidelberg.