CONSIDERING STRUCTURAL AND VOCABULARY HETEROGENEITY IN XML QUERY: FPTPQ AND HOLISTIC EVALUATION.

Brice Nguefack[1], Maurice Tchoupé Tchendji[2] and Thomas Djotio Ndie[3]

[1&2]Department of Mathematics and Computer Science, University of Dschang, Dschang, Cameroon
[3]National Advanced School of Engineering, University of Yaoundé I, Yaoundé, Cameroun

Abstract

The integration of XML data sources which have different schemas/DTD can originate structural and vocabular heterogeneity. In this context, it is difficult to write satisfiable queries. As a solution, many Information Systems focus on building approximate evaluation techniques for exact queries. As a project, we build flexible and preference XML query languages and associated evaluation algorithms. In this paper, we propose the Flexible Preference Tree Pattern Query (FPTPQ), a new TPQ that allows multiple items/names (resp. paths) for the same node, in order to integrate (resp. to locate) all the different instances of the database nodes. The FPTPQ enable to have preference nodes and ordering operators among label items and paths. We also provide a holistic algorithm that evaluates the FPTPQ and capitalises the preferences to determine the best available solutions. Illustrations and experimentations are realized to show the effectiveness of our solutions.

Keywords

XML Query with structural preference, Structural heterogeneity, vocabular heterogeneity, flexible query, holistic matching algorithm, ranked results.

1. Introduction

XML has become the standard format for information representation and data exchange among different systems. For interoperability needs, data of various sources which have been modelled independently, can be merged or exploited simultaneously. Generally, the integration of XML data that come from different sources may cause heterogeneity problems [1]. Heterogeneity in the domain of databases have different appreciations [2, 3]. Amount the types of heterogeneity, we denote structural heterogeneity and vocabular heterogeneity. The first refers to the fact that the same information instances may have different paths whose name and number of the nested database tree nodes (tag) are different. The second signifies that the instances of the same object (tag, attribute, text value) are represented with different names, that can be synonyms (e.g. skill, competency and expertise), abbreviations (course, crse), or any other group of similar words (hours and times, football and soccer). This problem has serious impact on the querying process. As a matter of fact, during the exact matching of exact queries, since all the constrains are considered as filters, a single erroneous object name among numerous ones means total failure,

DOI:10.5121/ijdms.2021.13601

The consideration of only one denomination among all the object instance names or paths may result to an incomplete set of solutions. For example, Let's consider the merging of the documents uwm.xml and wsu.xml available on [4] which contain respectively the schedule of the University of Wisconsin-Milwaukee (UWM) and the Washington State University (WSU) courses. Table 1 shows fragments of each of these documents. Even though these documents have been built for the same goal and for the same subject (courses scheduling), some information instances are contained in different tag (with similar names) or/and are located with different paths. The correspondent of the uwm.xml tags *course*, *course_listing* and *hours* are respectively *crs, course* and *times* in wsu.xml. The starting time of a course is located by the path *root/course_listing/section_listing/start* in uwm.xml and by *root/course/time/start* in wsu.xml. The integration of these two databases creates structural heterogeneity and vocabular heterogeneity. A non-aware user who needs the title of all the courses may have as result only the ones of *WSU* courses if he uses the tag course without considering *course_listing*. He will face the same consequences if all the different paths of the needed database node are not considered in the query.

Table 1. Fragment of the XML files of the two USA university course schedule, available on [4].

a) Fragment of uwm.xml

```
<root>
 <course_listing>
   <note></note>
   <course>400-327</course>
   <title>CONTEMPORARY FRENCH …</title>
   <credits>3</credits>
   <level>U</level>
   <restrictions>; (HU) PREREQ: FRENCH 303(215)
…</restrictions>
   <section_listing>
      <section_note></section_note>
      <section>Se 001</section>
      <days>TR</days>
      <hours>
         <start>12:30pm</start>
         <end>1:45pm </end>
      </hours>
      <bldg_and_rm>
          <bldg>CRT</bldg>
          <rm>B13 </rm>
       </bldg_and_rm>
       <instructor>Alkhas Alkhas</instructor>
   </section_listing>
</course_listing>
…
</root>
```

b) Fragment of wsu.xml

```
<root>
  <course>
    <footnote></footnote>
    <sln>10637</sln>
    <prefix>ACCTG</prefix>
    <crs>230</crs>
    <lab></lab>
    <sect>01</sect>
    <title>INT FIN ACCT</title>
    <credit>3.0</credit>
    <days>TU,TH</days>
    <times>
       <start>7:45</start>
       <end>9</end>
    </times>
    <place>
       <bldg>TODD</bldg>
       <room>230</room>
    </place>
    <instructor>B.
MCELDOWNEY</instructor>
    <limit>0112</limit>
    <enrolled>0108</enrolled>
  </course>
…
</root>
```

The characterization of the searched information alongside with the query evaluation process most consider these issues in order to find the complete set of solutions available for the user in the database. In the literature, some authors focus on tailoring a homogeneous database from the heterogeneous XML document collection. Other authors make some mapping between data sources schemas, that help to construct from each source document (resp query), a target one which is more appropriated for the querying process [5] [6]. In [7], AlHamad et al. produce a global schema for the entire database, alongside with mapping procedures between it and the multiple sources schemas, in order to provide a homogeneous view over heterogeneous XML data

[8], [9]. Further works [10], use ontology, web semantic [11], thesaurus, passed user experiences and other artificial intelligence techniques to automatically rewrite query. The purpose is to obtain from the user query, a more suitable one, with the respect of the database structure/format [12]. The resulting queries are more likely to produce solutions than the user one, since the user may not be familiar to database schema and vocabulary. These techniques are used in mediation systems. A rewriting system can produce as much targets queries as there are data sources or clusters with different schemas. Consequently, multiple evaluations are needed. The flexibility features are used after the formulation of the query by the user. The cited works do not allow the user to propose himself, some additional label items that may be useful to determine alternative solutions, or the complete set of the needed solutions. In fact, the existing query languages do not allow to have multiple paths to localized the occurrences of the same object, or multiple (similar) words as a node label, where each word can be one of the names used for the same database node. Moreover, the user may have in mind some potential related substitute words or objects which can help the evaluation process to also select alternative solutions in case the ideal query (query that is supposed to give the user a maximum satisfaction) does not have database images. But he will still be obliged to write other queries by continuously make several adjustments from the initial one, replacing some labels by others, sometime without changing the initial query organization. If this situation is more common in e-commerce context, it is a general problem for database querying system. Note that, as an XML document has a tree shape, XML queries that are written with the most popular languages XPath and XQuery can be represented as a combination of one or many trees, called Tree Pattern Queries (TPQ).

Different types of tree pattern query exist in the literature [13]. The most expressive ones enable the utilization of the wildcard * to match any single database node, or the Ancestor-Descendant (//) operators to allow the matching of the same object occurrences that have different paths. As example, If the user wants to select the titles of books and articles in the database dblp, he may use the wildcard "*" to have the paths dblp/*/title. To select all the start time of the courses in *merged_wsu-uwm.xml,* he may use *root//start.* But these operators are responsible of many useless solutions and does not enable any preference order. dblp/*/title will select the title of all the documents (articles, books, but also improceedings, master thesis and PhD thesis). *root//start* will select all the occurrences of the node *start* which are the descendant of the *root,* no matter what is in-between. Rather than using the *wildcard* and the *A-D* operators, a list of additional elements can be added as replacement items to query node labels/paths. This type of flexibility ensures that only the needed solutions are returned to the user. In some case, the replacement items can allow the selection of alternative solutions that are closed to user needs (example room if studio are not available). Such tree pattern is not useful only for the user. It can also be used to represent the results of a query reformulation [14] [15] [16] by a mediator system.

Flexibilities have been imported in database query process to write soft queries and allow more solutions possibilities. The concept of preference query, also none as bipolar query, is used to write query that have two parts: a first part has "must be satisfied" constraints, and a preference part is made of soft constraints. Contrary to the first part whose constraints are considered as filter (conditions whose every solution must absolutely satisfy), the satisfaction of the preference part is optional, but it enhances the correspondent solution value. i.e., the solutions which also satisfy the second part are must likely to be preferred by the user than those who satisfy only the obligatory constraints. None of the existing TPQ, even preference one, allow the integration of difference instance names for the same object, as node label. None of them allow multiples paths for the same query leaf node. In this paper, we propose (in section 3) a more general Tree Pattern Query called Flexible Preference Tree Pattern Query (FPTPQ). The FPTPQ enable replacement

items for node labels, and multiple paths to locate the multiple instances of the same database object. The items can be of equal value or classified in preference order using ordering operators, when the replacement items are the attributes of alternative solutions that have difference preference values, according to the user. For example, a user who need a room can add studio as alternative. Attributes of room will be more rated than the attributes of studios. The FPTPQ improves the preference operator of the language *prefSXPath,* proposed in our preview article [17]. For the evaluation of the FPTPQ, we proposed (section 4.4), the holistic matching algorithm FlexPrefTreeMatch which is an improved version of TreeMatch [18]. The matching of the query is paired with the calculation of each solution weight, utilized to determine the best solutions. Illustrations (Section 5) and experimentations (section 6) are made, in order to show the effectiveness of the FPTPQ and the algorithm FlexPrefTreeMatch: that is to show how the complete solution set is returned, and how the useless solutions caused by * and // are avoided.

2. Preliminaries

In this section, we define the concepts related to XML databases querying process, that are useful to understand this work.

2.1. XML and heterogeneous database.

An XML database is a collection of XML documents. An XML document [19] Consists of a set of hierarchical tags that describe the data they contain. An XML document must be well-formed, that is it must be in accordance with XML recommendations. The structure of XML documents is fundamentally tree oriented, so it can be modeled as a rooted tree $t = (\mathrm{N}, \mathrm{E})$ where N is a set of nodes labeled with the tag name for internal nodes, data or attributes for leaf nodes. E is the set of edges, each one represented as a couple $(n_i, n_j) \in (N \times N)$ that connect a node n_i, to n_j. DTDs or XML schemas when used, impose a structure/format to the XML database documents. They define the nature and the type of the elements that may be included in a valid document of the collection, and the way these elements are nested. An XML database can therefore be represented as a forest. XML documents validated against a DTD or XML Schema are said to be "Valid", The corresponding database object instances usually have uniform structure and same tag names, and can be considered as homogeneous. In the other hand, the integration of many XML data sources with different structure/format may result to a heterogeneous database. XML database heterogeneity can be interpreted in different way. In this paper, we are interested in vocabular heterogeneity and structural heterogeneity [20] . We talk of vocabular heterogeneity when the same object instances are expressed by different tag names, synonyms, abbreviations or other languages borrowed words. Structural heterogeneity refers to the fact that the same information instances may be located with more than one paths. Heterogeneity makes it difficult to write satisfiable query, using the common XML query languages. As XML document, XML queries can be modeled as a combination of trees, called Tree Pattern Queries.

2.2. XML query and Tree Pattern Query (TPQ)

To extract specified data from an XML database, many query languages have been developed. The most famous are XPath [21] and XQuery [22]. A common feature of these languages is a possibility to formulate paths in the database tree or forest. Such a path is a sequence of tree nodes from the root to the searched element occurrences. Regular expressions of XML query languages provide valuable methods for paths specifications based on XPath, and some formular to join the path images. XML queries can be translated into one or many trees, called Tree Pattern Query

[13] (TPQ) with the respect of the structure and the complexity of the query. Each TPQ is used to represent a useful fragment of the principal query. A TPQ is a tree $t_Q = (N_Q, E_Q)$ where N_Q is a set of nodes containing the root of t_Q and E_Q is the set of edges represented as couple $\left(n_{Q_i}, n_{Q_j}\right) \in \left(N_Q, E_Q\right)$ that connect the query nodes n_{Q_i} to n_{Q_j}. Several TPQ models exist in the literature with different kind of features, some are more expressive than others, according to the operators and relations they offer. One of the first tree pattern queries is the Tree Algebra for XML Tree Pattern Query (TAXTPQ) [23]. Its main features are the ancestor-descendant (A//D) and parent-child (P/C) relations [24]. If had brought the basics features for the other TPQ. The TAXTPQ is too rigid and the user has to master the structure of the database in order to adapt his request accordingly. The absent of only one edge or node in a potential solution tree prevents it from being in the final result of the matching, even if the candidate subtree is "almost perfect" [13]. As a response, Chen et al proposed the Generalized Tree Pattern (GTP) [25], a TPQ that enable some edges or nodes to be optional, by associating a mandatory/optional status to them, in other to increase the possible matched subtrees in the database tree. More than a limitation Through the Annotated Pattern Tree, Paparisos et al [26] allow the addition of a specification to an edge (*u, v*), which specifies how many matches to node *v* are to be obtained from each match with node *u*. Moreover, Lu et al [18] have proposed the Extended XML tree pattern that enable more relaxation with the wildcard node "*". The wildcard can match any single tree element. It is usually used when the associate's element is unknow or is not important. The risk with this operator is the abundancy of the corresponding matching element alongside with the query solutions. In fact, it uses to originate many useless solutions that imply costly filtering.

Sometimes, the user or the potential mediator system could have some additional elements that may reinforce the satisfiability of the query, when there is no assurance that the first (principal) searched element will be available. In this context, rather than using the wildcard and give a totally freedom to the matching algorithm to select anything, it is better to insert the additional elements as replacement items that will help to produce alternatives solutions which are near to the user initial needs. But the existing tree pattern does not allow it. The optional operator "*?*" is a beginning of a solution, but allow only an element to be optional [26]. With the logical operator OR and XOR, Izazi et al [27] allow the selection of one of two proposed sub path that finished at different leaf nodes. They don't integrate the fact that only some internal nodes/paths may need some replacement items. For example, if we consider the two documents of Table 1, we see that the starting *time* of the two documents courses are inside the tag *start*, while the parent tags are different, *hours* for the first document, and *times* for the second. The TPQ of Izazi et al does not allow to have *hours* and *times* as items of the same node.

2.3. Exact query, Flexibles and preference queries languages

Exact query is considered as query where all the constraints most obligatory be satisfied. All the constraints are considered as filter. The none satisfaction of all the conditions by a solution disqualify it from being in the final result set. Flexible queries are relaxed, allow soft matching and favour more solutions possibilities. Queries containing the operators "*?*" and "***" can be consider as flexible queries, since the first accept the fact that an element of the image tree can be absent and the second can be matched by any single database node (when different database nodes can be the image of the same query node). The wildcard is responsible of big solution set. At the end of the matching, all the solutions of such queries are considered to be of equal weight and are returned to the users without any ranking, obviously with all the useless solutions.

Preference queries, more that flexible queries also include operators useful for the calculation of the best solutions. As example we consider a tourist looking for a room in a luxury hotel with swimming pool and beach. Even if hotels with beach are difficult to find, they are more likely to be preferred by the users. If it is possible to consider the beach as optional (in flexible query), in the final query solution set, all the hotels room's will be returned in a random order, with no consideration of if they have a beach or not. The best formulation of that query is "I want a room in a luxury hotel, with a preference for hotels that have a swimming pool and a beach". In this case, it is better to use a preference query language [28] [29] [30] or the bipolar queries languages of our previews works [17] [31] [32] which make it possible to write queries with two parts: a first part containing the obligatory constraints that most necessarily be satisfied and a second part containing the elements of preferences whose availability magnify the corresponding solutions. At the end of the matching, the solutions that are not dominated, those which satisfy all the obligatory constraints and incorporate more elements of preferences than the others, are returned. Pareto's dominance concept through the skyline [33] [34] [35] [36] operator is commonly used to compare the solutions. But more is still to be done to make XML queries languages and TPQ more flexible even though they integrate preferences. The non-existence of database schema may induce the representation of an information instance by different words, in the same collection of documents. Moreover, in the context of heterogenous database, XML query language must allow the integration of all the different representations of an information in the query (for focused flexibility) while minimizing the utilization of the operators * and // which cause huge number of useless solutions.

3. Flexibles Preferences Tree Pattern Queries (FPTPQ)

3.1. motivations

Vocabular heterogeneity and structural heterogeneity make difficult to write query that capture all the different representations of the database tags. Several issues are considered:

Issue 1: The database mays contain some words (tags) and their synonyms, abbreviations, other language borrowed word. For example, in the database of a country labor's ministry, a *football club* can be considered as an *enterprise*, whereas in the database of the Football League the word used for the corresponding tag is "*club*"; For those who are used to American language, the collective sport where the ball is moved only with foot is called *soccer* rather than *football* in European country. Another example is the utilization of the words *option* and *specialty* to indicate the area of study. The same object instance may have different paths in the database. As example *root/course_listing/section_listing/start* and *root/course/time/start* in *merged_wsu-uwm.xml.* In this issue, the listed elements are similar, and thus have the same values. We only need to precise the different word/path instances no matter the order. The purpose is to select all the available solutions. Here we introduce the expression **Flexible Node (FN).** A FN is a node which have more than one label item. Its label items are separated by the operator "|". In the query S2 of **Figure 1**, the node whose label is *(country | location),* is a flexible node. The purpose is to maximize the satisfiability of the constrains associated to that node.

Issue 2: This issue concerns user preference node label which has multiple representations or paths in the database. Here, the objective is to maximize the satisfiability of the preference constraints. The list of the similar elements should be listed alongside with the preference operator. The associated node is called a **Flexible Preference Node (FPN).** A FPN is a preference node which has more than one label. The preference operator remains "!" as proposed in ours preview work [17]. In the last query of Figure 1, the node whose label is *(proc1|proc2)!,* is a FPN.

Issue 3: In some circumstances as e-commerce, the attributes (constraints) for alternative solutions can be added as replacement items alongside with preference order, in the same query. So that, if the ideal (user first choice) solution is missing, alternatives ones will be selected. The established preference order is used to calculate the best solutions among all the available ones. For example, in the last query, the node whose label is ***(Proc1 | proc2)>!*** is an Ordered Flexible Preference Node (OFPN). In an OFPN, the node label items are classified in ascending or in decreasing order, thus ***(Proc1 | proc2)>!*** Can also be written as ***(Proc2 | proc1)<!.*** In this node, the label item proc1 is more preferred than *proc2*. The solutions whose tree has *Proc1* will have a greater preference value than those whose tree has *proc2*.

The different types of nodes can be combined inside the same query.

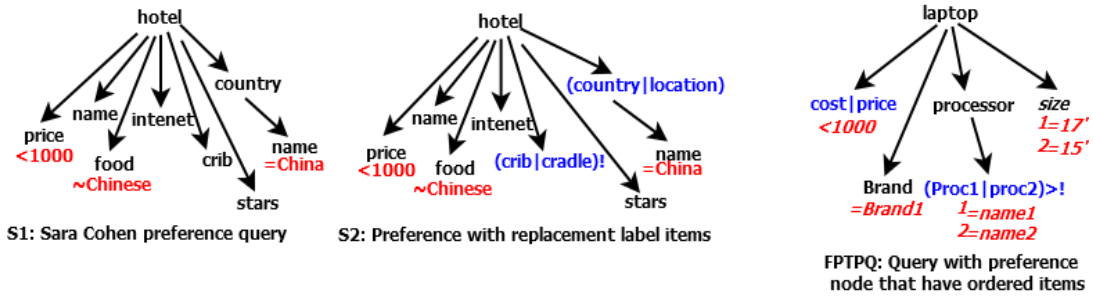

Figure 1. Sara Cohen preferences Vs flexible preferences

3.2. Flexibles Preferences Tree Pattern Queries (FPTPQ): language.

It is already possible to propose many preference alternatives for data values (processors name, screen size, etc.) using Sara Cohen preference language [30]. None of the existing TPQ allow to do so for internal node labels. In this section, we present the Flexible Preference Tree Pattern Query (FPTPQ), a model of Tree Pattern Query which gives the possibility of proposing many paths or/and many label items (words) for some query node, alongside with on-demand ordering operators which are useful for the calculation of solutions preference weights. This is another way to enable more flexibility at the level of the query writing module. Since the user can be the one to propose replacement items, the associated results are likely to satisfy him and save the querying systems from multiples query execution and useless filtering operations. To express preferences inside the query, we extend the *prefSXpaths* language proposed in our previews work[16]. As another example, let's consider the queries of Figure 1, from Sara Cohen et al [30]: In S1, a tourist (Sam) needs a hotel to stay at when attending a conference in China. Ideally, Sam would like a cheap (at most 1000 RMB per night) hotel in China. Since he would like to taste the local food, he would like Chinese food to be served at the hotel. Sam needs an Internet connection, to keep in touch. Finally, Sam will be bringing his wife and new baby, and so will need a crib. Despite it is represented as a preference query type, in a heterogeneous database, the *crib* can be expressed by its synonym *cradle.* In the place of the tag *country*, the database designer could have used the word *location*. In order to maximize the satisfiability, synonyms or equivalent word (*cradle* and *location*) can be added as replacement item for the label *country* and *crib* in order to have query S2. In some cases, the main label item, better contribute for user satisfaction than the other replacement items.

In a preference query, a priority/preference order can be set among the replacement items of preference nodes. In this case, the operators "<" (resp ">") are added before "*!"* to indicate that the replacement item are listed in ascending (resp in decreasing) order of preference.

Figure 2-b show how to represent a FPTPQ (of Figure 2-a) that has FN and FPN, using their associated variable. Figure 2-d do the same for the FPTPQ of Figure 2-c which has both FN and OFPN. All the other nodes constraints are expressed as in the Extended Tree pattern query [18]. The preference operator still remains **"!"**. We can now have preferences nodes that have replacement items. The binary operator "|" is used as separator of multiple paths or label items.

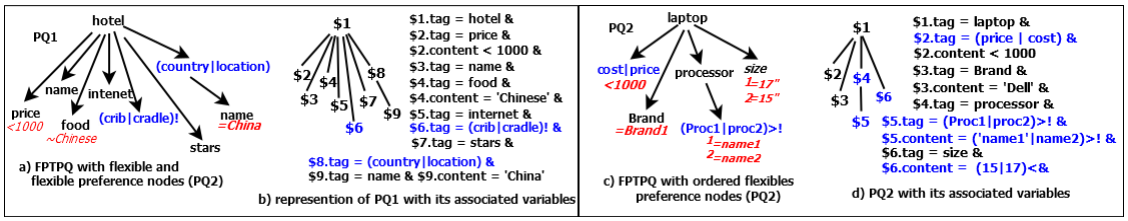

Figure 2. Expressing the FPTPQ with variables.

As label of a preference query node, $(p_1|p_2|\ldots|p_n)!$ means that in the absence of P_1, the items $P_2,\ldots,p_n$ will equally replace it. Figure 3-a show the tree representation of the query ***A[(D|E)!/F]/B!/C.*** In this query, *node 2* is a preference node, *D* and *E* are both label items of a preference node, with the same weight. In the absence of *D*, *E* will equally be considered. In the query of Figure 3-**b**, the preferences nodes items are ordered. This means there are classify in increase order like ***(C|D)<!*** or in decreasing order like ***(G|H)>!***. Here, the label item *D* is more valuable than *C*. The solutions whose tree carry *D* will be more valuable than those whose tree carry *C*. In the query of Figure 3-b**,** the item *H* can replace *G*, but with a lower preference value.

3.3. Assignation of preferences values to preference node items

We need to know that preferences values are assigned to preferences node items, according to the type of preference node there are associated to. All the items of FPN are assigned the default preference value which is "**1**" like in the example of Figure 3-**a**. For OFPN that carry a lower operator (<), the label items from the first, are assigned preference values respectively from the integer 1 to **N**, where **N** is the number of items. When the preference node has the operator ">", preference values are assigned to its label items from "**N"** to "**1"**. In Figure 3-**b**, the node 3 has the operator "<" which mean its label item are classify in ascending order. So, the label item C has "**1**" as preference value and the label item D has "**2**" as preference value. For the node 6, the label items are classified in descending order; its values then start from "***2"*** for *G,* to "**1**" for *H*.

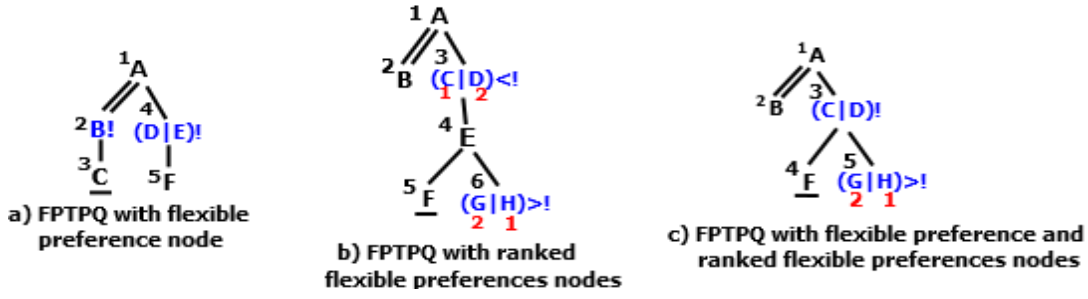

Figure 3. FPTPQ with different types of preference nodes

4. Evaluation of the FPTPQ: FlexPrefTreeMatch

In this section, we present an evaluation approach of the FPTPQ. The purpose is to minimize the FPTPQ by removing all the labels items that do not appear in the database, then the minimized

FPTPQ is matched with an extended Dewey based index using a holistic algorithm that is based on treeMatch [18]. During the matching, the preference value of each solution is also calculated.

4.1. The algorithm TreeMatch

We used the algorithm TreeMatch as the backbone of our proposed algorithms because it is able to optimally (in term of I/O complexity) processed the Extended Tree Pattern Query, which is one of the most featured and flexible TPQ, since it allows negation function, order-based axis and wildcards. In fact, treeMatch has one of the larger optimality classes in terms of input-output. The optimality class of a tree pattern matching algorithm represent the set of TPQ it is capable of optimally evaluate. Since the FPTPQ integrate replacement items, preference operator, and ordering among items, some modifications have been made on TreeMatch for its evaluation. The evaluation process of a FPTPQ start with some precomputing which consist of the minimization of the FPTPQ label items, node's identification, attribution of preference values to each node item, the determination of all the paths associated to each query leaf node, alongside with their corresponding preference values.

4.2. Precomputing: FPTPQ minimization, node identification and calculation of preference values of all paths.

Before the evaluation, the FPTPQ label item are minimized, using the database tag names list. In fact, all the label items that do not appear in the database tag list are removed from the FPTPQ. the FPTPQ query nodes are then numbered, using integer values that are used to identify them.

Because a node label can have many items, it is not appropriated to identify them with their label or with one of their label items, since each item may be manipulated separately. For example, if a node label is "A|B", A|B or A are not more appropriate identifier for it, since A|B is too long and each database index list contains the occurrences of only one object. Integer identifiers are associated to the node with the respect of their position. The Figure 3 show how the nodes of different types of queries are numbered. The couple (i, lj) is used to refers to the label item (name) l_j of the node whose id is j. During the evaluation process, the preference value of each solution is progressively calculated using the preferences values associated to each query branching node that are stored in the preference locate match table. The description of this table is shown in section 4.3, alongside with the procedure that need it.

4.3. Used Data structures

The inputs of the algorithm treeMatch are a *FPTPQ* and the lists T_q associated to each label item q of the minimized query. T_q contains the extended Dewey label of all the tag occurrences whose name is q. e_q is used to refer an element of a T_q list. $Cur(T_q)$ is used to denoted the current element pointed by the cursor of T_q. The procedure $advance(T_q)$ is used to advance the cursor of T_q to the next element. Like for the algorithm TreeMatch in [18], a set S_i is associated to each query branching node. Here, "i" no more the branching node label, but its id. Each element e_q of the set is a triplet (label, intVector, outputList) where *label* is the extended Dewey label of e_q. *intVector* is a vector of integer whose size is equal to the number of descendants of q. Compared to *bitVector* used by *TreeMatch*, *intVector* has many rules. Its first rule is to tell whether e_q has the proper children or descendant with the respect of the query (as *bitVector* for TreeMatch). Its second function is to save the current (partial) preference value of the potential solutions. Indeed, each integer of intVector represent the preference value of all the its associated descendant subtree. Given a child node q_c of q, let consider $intVector(e_q)[q_c]= V$: $V>=0$ if and only if there is a

database element e_{qc} such as e_q and e_{qc} satisfy the query relationship between q and q_c and V is the preference value of the subtree rooted by q_c; $V = -1$ if not. *OutputList* contains the elements that potentially contribute to final query answers.

4.4.Algorithms flexPrefTreeMatch and associated procedures and functions.

4.4.1. The principal algorithm: flexPrefTreeMatch

Line 1 locate the first match label of each query leaf node. If a leaf node is a FN then all its label items first matches would be located and the one with the minimum match label will be first processed. Now the function *prefGetNext* select among all the query leaf nodes, the one which is going to be processed (the one that has the item with the minimum match label). *prefGetNext* return a couple (f_{id}, f_{act}) where f_{id} is the *id* of the next leaf node to be processed, and f_{act} is the label item of f_{id}, that has the minimum current *matchLabel*. The purpose of line 4 and 5 is to insert the potential matching element inside the outputList of $NDB(f_{id})$. After the treatment of f_{id}, the cursor of T_{fact} is advanced to the next element. Line 7 update the set encoding and line 8 locate the next matching element to individual root to leaf path. Finally, line 9 do the appropriate update for the final solutions; for FPTPQ which has preferences, the weighted solutions table is constructed.

Algorithme1: flexPrefTreeMatch

```
1: PrefLocateMatchLabel(Q);
2: while (¬end(root)) do
3:     (fid, fact) = prefGetNext(topBranchingNode);
4:     if (fid is a return node)
5:         addToOutputList(NAB(fid), cur(Tfact));
6:     Advance (Tfact); // read the next element in Tfact
7:     prefUpdateSet(fid, fact ); // update set encoding
8:     prefLocateMatchLabel(Q); // locate the next element with matching path
9: emptyAllSets(root);
```

4.4.2. PrefGetNext and other procedures and functions used by flexPrefTreeMatch

The index used by flexPrefTreeMatch remains the T_q lists, where each T_q list contains the Extended Dewey label of all the database occurrences of q. A list T_q is visited only if q is the label item of a query leaf node. Initially a pointer is positioned at the first element of T_q. *PrefLocateMatchLabel* is a very important procedure, its purpose is to locate the first elements whose path match one of the individual root-leaf query paths with the respect. In spite, for FPTPQ, many paths can be associated to a query leaf node. The preference locate match table is used to associate to each query leaf id, all its corresponding root to leaf paths in other to facilitate the matching. Table 2 associate to each leaf node item of the query of Figure 5-b all its corresponding root to leaf paths. For leaf node which are flexibles, all its label replacement items have to be considered during the matching. Let consider a leaf node whose label items are $q_1..q_n$; during the matching, if the current matchLabels of these items are respectively $e_1..e_n$, then, the label with the minimum (by lexicographical order) will be first selected to make sure that the evaluation is being made by lexicographical order.

The function *minMatchLabel(n)* return the minimum of the current matchLabels of all the label items of the node n. The function *minItem (n)* return amount the label item of n, the one that has the minimum current match label. That is $n_{imin} \mid cur\ (Tn_{imin}) = minMatchLabel(n)$.

Give the current matchLabel *e* of the node label *n*, *flexMB (n, b)* return all the matchLabel of n that cover (are ancestors or parent of) e. *flexMB* help to make sure that if many occurrences of the branching node (b) carry e, we always start to process the deepest one, since the evaluation is bottom-up. The function *minValue(v)* return the minimal value of the *intVector v*. The function *prefValue($e_{q'}$, e_q)*, return the preference value of the path that link $e_{q'}$ to e_q. It is equal to the preference value of e_q added to the sum of all preference node value between $e_{q'}$ and e_q. This function is used by the procedure *updateAncestorSet* to set the integer value of the nearest ancestor branching node set. Assume that q is a branching node and q_i is it children, *intVector (e_q, e_{qi}) = prefValue(e_q, e_{qi}).* Like the matching process, the calculation of the final solution preference value (weight) of a node is bottom-up. This mean that, during the matching, the preference value of a branching node matchLabel is equal to the sum of all the preference values of its children subtree images. The function flexSatisfyTreePattern(e_{qi}, e_q) test whether the document element e_{qi} is covered by the branching node matchLabel e_q; it return true if *intVector (e_q)[e_{qi}]>=0*;

Function prefGetNext(n)

1: **if** (isLeaf(n) then
2: return $(n, n_{min})| \, n_{min} = \text{minItem(n)}$
3: **else**
4: **for each** $n_i \in$ NDB(n) **do**
5: (m_i, f_i) = prefGetNext(n_i)
6: **if** (isBranching(n_i) and ¬empty(S_{ni})
7: return (m_i, fi)
8: **else** e_i = max{p | p ϵ flexMB(n_i, n)};
9: **end for**
10: max = maxArg_i {e_i};
11: **for each** $n_i \in$ NDB(n) **do**
12: **if** $(\forall e \in \text{flexMB}(n_i, \text{n}): \text{e} \notin \text{ancestors}(e_{max}))$
13: return (m_i, f_i);
14: **endif**
15: **end for**
16: $min = \text{minarg}_i \{f_i \,|f_i \text{ is not a return node}\}$;
17: **for each** e ϵ flexMB(n_{min}; n)
18: **if** (e ϵ ancestors(e_{max}) updateSet(S_n; e);
19: **end for**
20: return (m_i, f_{min});
21: **end if**

Function *flexMB* (n, b)

1: **if** (isBranching(n)) **then**
2: Let e be the maximal element in set S_n
3: **else** Let e = minMatchLabel(n);
4: **Return** a set of elements **a** that is an ancestor of **e** such that **a** can match node **b** item in the path solution of **e** to path pattern $\mathbf{p_n}$

Function prefSatisfyTreePattern(e_{qi}, e_q)
1: **if** (intVector(e_q, e_{qi}) >=0) return true;
2: **else** return false;

Procedure prefUpdateSet(q; e)
1: prefCleanSet(q, e);
2: add e to set S_q; *//set the proper intVector(e) using the locate match table.*
3: **if** (¬isRoot(q) ∧ (minValue(intVector(e))>= 0)) **then**
prefUpdateAncestorSet(q);

Procedure addToOutputList(q, e_{qi})
1: **for each** $e_q \in S_q$ **do**
2: **if** (prefSatisfyTreePattern(e_{qi}, e_q)) **then** outputList(e_q). add (e_{qi});

Procedure prefCleanSet (q; e)
1: **for each** element $e_q \in S_q$ **do**
2: **if** (prefSatisfyTreePattern(e_q,e))
3: **if** (q is a return node)
4: addToOutputList(NAB(q), e);
5: **if** (isTopBranching(q)
6: **if** (there is only one element in S_q)
7: output all elements in outputList(e_q);
8: **else** from the set S_q construct the weighted solution table
9: delete e_q from set S_q;

Procedure prefUpdateAncestorSet(q)
1: /*assume that q′ = NAB(q)*/
2: **for each** $e \in S_{q'}$ **do**
3: **if** (intVector (e, q) = -1) **then**
4: intVector(e, q) = **prefValue(e, q);**
5: **if** (¬isRoot(q) ∧ (**minValue(intVector(e))**>= 0))
6: prefUpdateAncestorSet(q');

4.4.3. Computation of the best results

After the matching by the function *flexPrefTreeMatch*, the *weighted solution table* is constructed using the set of the top branching node. The *weighted solution table* contains only the solutions that integrate at least the obligatory constraints. It associates to each solution its corresponding preference weight. The solutions are sorted by increase order of weight. In fact, the weight of every solution of the final *outputList* is the sum of all the integer (weight of the children subtree) of its corresponding *intVector*. The solutions are inserted in the table with the respect of theirs weight values in order to avoid a sorting operation, so that at the end of the insertion, the table is already sorted. The top-K best solutions (with the highest preference values) are returned to the user. The next section shows some illustrations of different FPTPQ execution process.

5. Illustration of the evaluation of FPTPQ by treeMatch

5.1.Illustration of flexPrefTreeMatch for queries with flexible nodes.

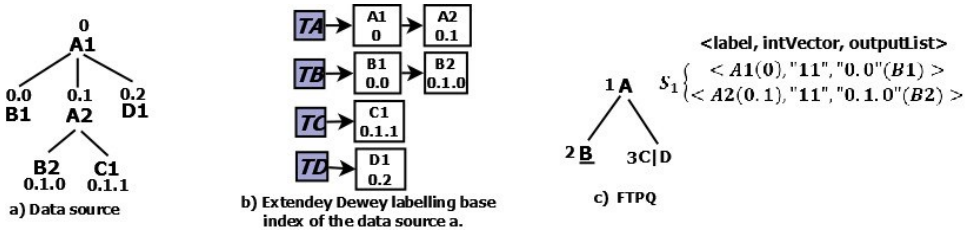

Figure 4: Illustration: evaluation of a FPTPQ that has a flexible node

Initially, the procedure *PrefLocateMatchLabel* locate *B1* for the node 2 and for the *node 3*, *C1* and *D1 are located, but C1* is selected first because it has the minimum matchLabel. The function prefGetNext return (3, C) because C1 is deeper than B1. Then A2 is added in S_1 with the intVector "*-10"* to indicate that at this moment, the second child of A2 has been found. The cursor of T_A is advanced. At the next stage, B1 and D1 are read by PreflocateMatchLabel and *(2, B)* is returned by PrefGetNext, A1 is added in the set with the bit vector "0-1" and outputList <0.0> (since B is a return node); the cursor of T_B is advanced. Early in stage 3, *B2* and *D1* are read and PrefGetNext return *(2, B)* because B2 is the deepest and has the minimum matchLabel. B is also a return node, so B2 is added in the outputList of its corresponding matchLabel ancestor element (A2). The intVector of A2 is updated from "-10" to "00". The last element to be read is D1, and the set S_1 is updated. That is the *intVector* of A is update from "0-1" to "*00"*. The query does not have any preference node, as the label items of the same flexible node, *C* and *D* are of equal values. A node (of id -1) is created to merges all the outputList of the set S_1: $S_{-1} = \{<-1, "0", (0.0, 0.1.0)>\}$. The set {B1(0.0), B2(0.1.0)} is returned. Notice that *B1* is also returned only because of a replacement label item (*D*) has been added to the FPTPQ for the node 3. Using the Extended Tree Pattern, the user may have been partially satisfied or would have been obliged to write and execute almost the same query, replacing C with D as the label of node 3.

5.2. Illustration of flexPrefTreeMatch: evaluation of queries with ordered preference node label items.

The Figure 5-a shows an example of a FPTPQ which has two ordered flexible preference nodes, lets describe its evaluation process using the algorithms treeMatch. First of void the query nodes are numbered (Figure 5-d), the labels Id are minimized (since the item C does not exist in the database, it is deleted) and the preference locate match table of Table 2 is constructed from it. The sets S_1 and S_4 are associated respectively to the branching nodes 1 and 4. The intVectors in blue shows the preference weight values of each path, calculated from the leaf matchLabel to the associated branching node matchLabel. In the first stage, B1, F1 and H1 (H1 is selected before G1 because it has the minimum match Label by lexicographical order) are read by prefLocateMatchLabel. A1 is added to the set S_1 with *"0.0.0"* (matchLabel of B1) in his outputList, since B is the return node of the query. At this moment A1, only the first child of A1 has been read, Its corresponding intVector is "0,-1". E1 is added to the set S_4 with the intVector "0,1" whose guarantee that E1 matches all its corresponding subtree, since E1 carries two children F1 which is not a preference node item, and H1 whose preference value is 1. Therefore, the intVector A1 (the corresponding NAB matchLabel) is updated to "0,3". The integer 3 is the preference value of the subtree rooted by D1. Later, F2 is read and it corresponding NAB (E2) is inserted in S_4 with the intVector "0,-1". When F2 is read, intVector(E2) become "0,1" since E2 has his two children and the second is the item of the *preference node 6*, whose preference weight is equal to "1". Later when B2 is read, A2 is added to the set S_1 with the intVector "0,1" and B2 in its outputList. Afterwards F3, G1, B3 are read in this order, followed by the insertion of

"*0.2.2.0"* (matchLabel of E3) in S_4 and the insertion of "0.2" (matchLabel of A3) in S_1 respectively with the intVectors '0,2' and '0,4'. Then, F4 and B4 are read, E4 is added in S_1 with the intVector '0,0', even if it does not have a child G or H. The reason is that node 6 is a preference node, its satisfaction is not compulsory in a solution tree. A4 is added to S_1 with the intVector "0,0" since it does not carry any preference node item. Finally, B5 is read, and its *NAB A5* is added to S_1 with the intVector "*0,-1*". The execution is stopped, since all the elements of the input list associated to the query leaf label items have been read.

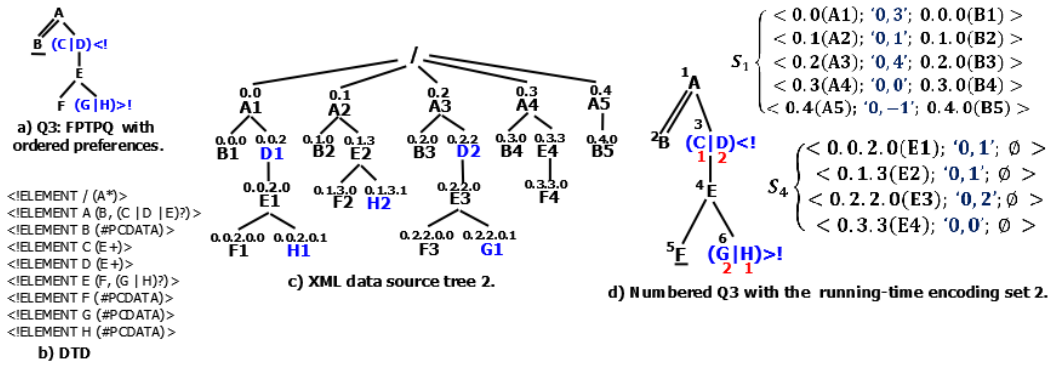

Figure 5. Illustration: evaluation of a FPTPQ with ordered preference node label items.

Table 2: Preference locates match table for the FPTPQ of Fig.6.

Leaf node id	Label items	Root to leaf paths	preference at the level of the NAB "1"	preference at the level of the NAB "4"
2	B	A/B	0	-
5	F	A/D/E/F	2	0
		A/E/F	0	0
	G	A/D/E/G	2 + 2	2
		A/E/G	0 + 2	2
6	H	A/D/E/H	2 + 1	1
		A/E/H	0 + 1	1
	E	A/D/E	2 + 0	0
		A/E	0 + 0	0

Table 3: Weighted solutions table for the example of Figure 5.

Solution id	name	weight	position
0.2.0.	B3	4	1
0.0.0	B1	3	2
0.1.0	B2	1	3
0.3.0	B4	0	4

At the end of the matching process, the weighted solutions table of Table 3. is constructed from the top branching node set S_1. We can see that the solution 0.2.0. (B3) has the greatest preference value and can then be considered as the best (most preferred) solutions. The solutions are printed per order of user preference. The possibility can be given to the user to precise the number (K) so that the top-K solutions will be returned.

With these illustration examples, we can show mayor differences between The FPTPQ and the other tree pattern queries. The replacement items enable more flexibility and more satisfiable results for query, than the existing most famous and featured TPQ like GTP [25] and the extended tree pattern query [18]. In fact, since they do not allow replacement items, the user would have written a query with only the first item of each flexible node. The consequence is that no solution would have been returned for the example of figure 3, since the database does not have any node (tag) named "*C*". Without the replacement items H and D, only the solution B4 which has the lowest solution weight would have been returned. Moreover, if node 3 was not a preference node,

B4 would not have been selected. Unless all the flexible nodes were replaced by the wildcard "*" which is responsible of a many useless solutions. This shows how the combination of replacement items and preference nodes contribute to ensure user satisfiability via the proposition of numerous solutions that remain close to the user needs.

Since none of the existing tree pattern query allow *flexibles nodes*, *flexible preference nodes* and *flexible ordered preference node* like describe in section 3.2. If We can assert that a new class of query have been created, the class of query which contain these three types of nodes, we cannot yet assert that flexPrefTreeMatch has a new optimality class. To do so, further analysis need to be done. Our main purpose was to propose the PFTPQ alongside with evaluation algorithms.

6. Experimentations: evaluation of FPTPQ with FlexPrefTreeMatch

In this section, we compare the evaluation results of the FPTPQ against the evaluation results of a TPQ written for the same need. We used the XML datasets DBLP (regular structure imposed by a DTD), Treebank (irregular, with no DTD), *uwm.xml* and *wsu.xml* which are two databases with different structures that was merged to produce *merged_wsu-uwm.xml* which suffer of structural and vocabular heterogeneity. These datasets are available on the University of Washington XML Data repository [4]. In fact, the FPTPQ help to express all the desired information set through replacement items, contrary to the other TPQ like the GTP [25] or the extended tree pattern query [18] whose execution may result to an empty or incomplete solution set. Moreover, these TPQ may originate many useless solutions through the utilization of the wildcard "*" and the A-D ("//") operator. This experiment is used none to compare algorithms execution times, but to show how the FPTPQ and FlexPrefTreeMatch may be effectively used to express queries which produce only or the complete set of desired solution, when the replacement items are well inserted.

The percentage (P) of useless solution is calculated based on the exact number of available solutions. Thus, if the number of returned useless solution is greater, P will be greater than 100%.

6.1. FPTPQ contribution for the searching of all the needed solutions.

Table 4. Comparison of FPTPQ (with appropriate replacement items) evaluation results with the evaluation results of classic TPQ.

N°	XML data (.xml)	Evaluation results of TPQ with only one label		Evaluation results of FPTPQ with appropriate replacement items		Number of lacking solutions	% of lacking solutions
		Tree Pattern Query (TPQ)	Number of solutions	FPTPQ with appropriate replacements items	Number of solutions		
1	**merged_wsu-uwm**	**TQ1:** root//credit	3924	**FQ1:** root//(credit|credits)	6036	**2112**	**34,99%**
2	**merged_wsu-uwm**	**TQ2:** root//sect	3924	**FQ2:** root//(sect|section)	8499	**4575**	**53,83%**
3	**merged_wsu-uwm**	**TQ3:** root/course_listing [//hours/start]//instructor	45 75	**FQ3:** root/(course_listing|course) [//hours/start]//instructor	8499	**3924**	**46,17%**
4	**merged_wsu-uwm**	**TQ4:** root/course/place/bldg	3924	**FQ4:** root/((course/place) |(course_listing//bldg_and_rm))/bldg	8499	**4575**	**53,83%**
5	**dblp**	**TQ5:** dblp/book/title	845	**FQ5:** dblp/(book|article)/title	112454	**111609**	**99,25%**
6	**dblp**	**TQ6:** dblp/book/author	1153	**FQ6:** dblp/(masterstheses|book)/author)	1158	**5**	**0,43%**
7	**dblp**	**TQ7:** dblp/masterstheses/author	5	**FQ6:** dblp/(masterstheses|book)/author)	1158	**1153**	**99,57%**

The FPTPQ through replacement items can help to completely express the user needs inside one query and therefore, reduce multiple query adjustment and execution. As query example, let's consider a user who wants to print the credit of all the courses present in the document merged_wsu_uwm.xml (the merged result of wsu.xml and uwm.xml). Because the occurrences of credit are represented inside two differences tags (credit and credits), the query TQ1 of Table 4 will produce only 3924 titles over the needed 6036 that are returned by the FPTPQ *FQ1*. The same issue is caused by the query TQ2 that print 3924 section nodes over the available 8499 that are completely returned with the execution of the FPTPQ *FQ2*.

Let consider now a user who want the titles of all the books and articles of dblp.xml. With the TPQ TQ5, only the titles of books (only 845 solutions over 112454 needed) will be returned, and the user will be obliged to write another query, replacing "book" by "article" in order to obtain the other 112454 titles. Only one execution is need with the query FQ5 to produce the complete set of solutions. The last column of Table 4 show that an appropriate utilisation of replacement items in a FPTPQ has helped to select more than *99*% of needed solutions that have not been selected with the utilization of the classic TPQ.

6.2. The FPTPQ help to avoid useless solutions caused by the wildcard "*"

During the evaluation of a TPQ, the wildcard can be matched by any single node. Consequently, it causes many useless intermediate results and unsatisfiable solutions. When the user is aware of all the words used to express all the instances of an object (document tag), it better to use them rather than "*". The Table 5 shows FPTPQ (with appropriate multiple label items) evaluations results compare to TPQ where "*" have been used in the place of the multiple items. In the document merged_wsu_uwm.xml, courses are now described inside two tags: *course* (from uwm.xml file) and *course_listing* (from wsu.xml file).

Table 5: Avoiding useless solutions with FPTPQ over TPQ with wildcard "*".

N°	XML database (.xml)	Execution of TPQ with the wildcard ("*")		Corresponding FPTPQ with appropriate replacement items.		Useless solutions caused by "*"	% of useless solutions caused by "*"
		TPQ with "*"	Number of solutions	FPTPQ with replacement items.	Number of solutions		
1	merged_wsu-uwm	**SQ1**: root/*[//hours/start]//instructor	8499	**FQ3:** root/(course_listing|course)[//hours/start]//instructor	8499	**0**	**0%**
2	merged_wsu-uwm	**SQ2**: root/*//*/bldg	8499	**FQ4**: root/((course/place)|(course_listing//bldg_and_rm))/bldg	8499	**0**	**0%**
3	merged_wsu-uwm	**SQ3**: dblp/*/title	328859	**FQ5**: dblp/(book|article)/title	112454	**216405**	**192%**
4	dblp	**SQ4**: dblp/*/author	716488	**FQ6**: dblp/(mastersthesis|book)/author	1158	**715330**	**61773%**
5	Treebank_e	**SQ5:** //PP[//VP/IN]/*/VBN	676	**FQ7:** //PP[//VP/IN]/(ADJP|VP)/VBN	96	**580**	**604%**
6	Treebank_e	**SQ6:** //*[//VP/IN]/*/VBN	28314	**FQ8:** //(PP|SBARQ)[//VP/IN]/(ADJP|VP)/VBN	89	**28 225**	**31713%**
7	Treebank_e	**SQ7:** //*[//VP/IN]/NP	435689	**FQ9:** //(ADJP|NP)[//VP/IN]/NP	98352	**337 337**	**343%**

Replacing the wildcard "*" of the query *SQ1* by these two tags names to obtain *FQ3* will originate only the needed solutions, since no other tag can be matched as a *course*. But the other tree pattern queries of Table 5. show how "*" caused overabundant useless solutions, even for simple queries. In the merged document, not only the courses can be scheduled in a building (bldg). Different sections of the same course can be scheduled in different building. With the operator "*", a user who need only the buildings of courses may be obliged to find between other useless ones (the building of section). Moreover, preference operator (*course_listing*|*course)<!,*

course|course_listing)>!) can be added to give a privilege to the buildings which are carried by the item (tag) "*course*". i.e., the building of the UWM courses. In *dblp* dataset, the cited documents can be articles, improceedings, master thesis, PhD thesis and books. The query *SQ4* is written using the extended tree pattern query for a user who wants only the titles of books and articles. "*" causes the selection 216405 (328859 - 112454) unneeded ones.

The utilisation of FPTPQ to write query *FQ5* (by replacing *"*"* with *book|article)* help the FlexPrefTreeMatch to return only the needed solutions (only the titles of books and articles). Queries *SQ5, SQ6, SQ7* produce a huge number of useless solutions. This is because of the heterogeneity of the dataset *Treebank*.

Table 6. Advantages of replacement items over the A-D operator

N°	XML database	Result of TPQ with A-D (//) Relations		Results of FPTPQ with replacement items in the place of "//"		Number of useless solutions caused by "//"	% of useless solutions caused by "//"
		Tree pattern query with "//"	Number of solutions	FPTPQ with needed replacements items	Number of solutions		
1	merged_ wsu-uwm	**AQ1:** root[//hours/start]// instructor	8499	**FQ10:** root/(course_listing|course) [//hours/start]// instructor	8499	**0**	**0%**
2	merged_ wsu-uwm	**AQ2:** root//bldg	8499	**FQ4:** root/((course/place)| (course_listing//bldg_and_rm))/bldg	8499	**0**	**0%**
3	dblp	**AQ3:** dblp//title	328859	**FQ5:** dblp/(book|article)/title	112454	**216 405**	**192%**
4	dblp	**AQ4:** dblp//author	716488	**FQ6:** dblp/(mastersthesis|book)/author)	1158	**715 330**	**61773%**
5	Treebank_e	**AQ5:** //S/VP//NP/NNP	55288	**FQ11:** //S/VP/(VP|PP)/NP/NNP	6518	**48 770**	**748%**
6	Treebank_e	**AQ6:** //PP[//VP/IN]//VBN	6262	**FQ7:** //PP[//VP/IN]/(ADJP|VP)/VBN	96	**6 166**	**6423%**

The chart of Figure 6 show the comparison result of the number of solutions returned by TPQ that have "*" and FPTPQ where "*" have been substituted by the appropriate replacement items. The FPTPQ contains only the needed and appropriate items. Line 1 and 2 of the table show that the number of solutions returned by the TPQ and the FPTPQ are equals. This is because the two needed nodes label "*course_listing"* and *"course"* are the only possible ones which can be the image of "*". The query *FQ11* shows how a FPTPQ can avoid more than *61773%* of useless solutions caused by "*" (if the TPQ *SQ4* is used) in highly irregular databases.

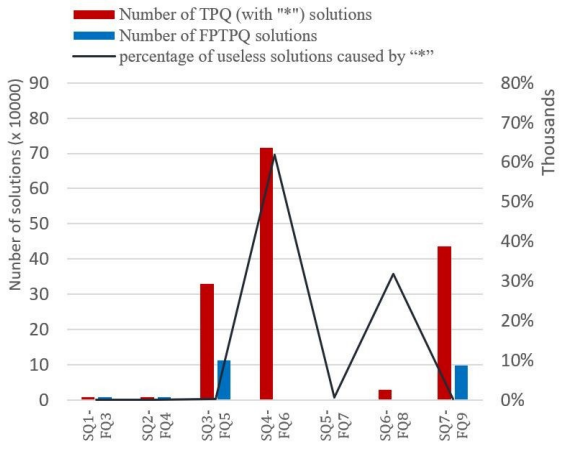

Figure 6. Showing how FPTPQ help to avoid useless solutions caused by "*".

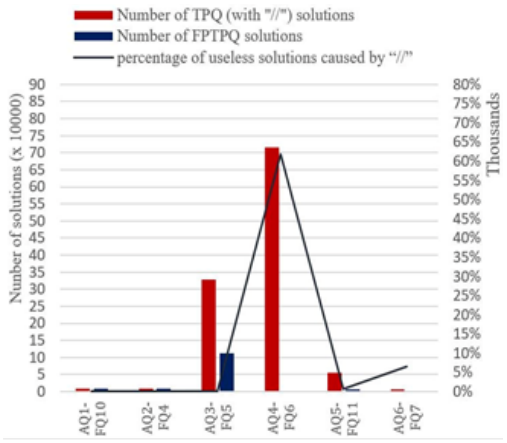

Figure 7. Showing how FPTPQ help to avoid useless solutions caused by "//".

6.3. Advantages of a proper utilization of replacement items in a FPTPQ rather than A-D operators.

When the schema is absent, a distance between XML nodes (number of node than are in-between) may not be known, then the utilization of the ancestor-descendant operator (//) is justified. But an inappropriate utilization of this operator causes the selection of solutions whose paths have different lengths and different nodes labels. The execution of a query like A//B imply the selection of A/B, A/*/B, A/*/*/B, …, where * can be anything. The execution of this operator produces many useless results linked to unnecessary paths. The Table 6 shows how useless intermediate solutions can be totally avoided when the replacement items are correctly inserted inside the FPTPQ. The execution of the query AQ3, AQ4, AQ5 and AQ6 produce a huge quantity of useless solutions. AQ4 induce 61773% (calculated base on the number of needed solution) of useless solutions. This huge quantity of useless solutions is due to the fact that all the database instance of the return node are read with not enough information to filter them. In fact, there are 328859 documents in *dblp* and the operator "*//*" of the query *AQ3* (resp *AQ4*) allows the selection of their title (resp of all the authors), no matter the type of document. When only the titles (resp authors) of books and articles are needed, it is preferable to use query *FQ5* (resp *FQ6*). The execution of *AQ1* and *AQ2*, do not generate useless solutions because all the instructors of the database are needed (the number of instructors needed is equal to the number of instructors replaced by //). The chart Figure 7 shows the higher percentage of useless solutions engender by *"//"*. We can see that four of the six queries produce over 150% of useless solutions. Query *AQ4* engendered more than 60000% of useless solutions. The consequence of such enormous quantity of useless solution is that it may confuse the user and make him abandon its searching.

To conclude this section, we can assert that a proper utilization of replacement items in a FPTPQ help to express all the needed solution, and avoid incomplete results that are caused by TPQ, since they do not allow multiple items (terms) to represent all the names of the difference tag instances. Even if the operators "*" and "*//*" remain useful, mainly when the database schema is unknown, they cause huge number of unsatisfiable solutions. When the replacements items are known and are added in the correct places of the FPTPQ, unsatisfiable solutions are totally avoided, and only the useful solution set is returned.

7. Conclusion

Obtaining satisfiable queries for XML databases that have structural and vocabular heterogeneity remain an important challenge. We proposed the Flexible Preference Tree Pattern Query (FPTPQ), a TPQ that allows to have multiple items as node label and multiples paths to locate the same query node, ordering and preference operators. The FPTPQ can be used in any XML database, to characterize in the same query the user both first choice solution and alternatives ones. The FPTPQ enhance the satisfiability of both preference and non-preference nodes. For the evaluations of FPTPQ queries, we proposed the holistic algorithm FlexPrefTreeMatch which match the FPTPQ with the database index based on extended Dewey labelling scheme, while calculating each solutions preference weight. Illustrations and experimentations verify the effectiveness of the FPTPQ and the correctness of the algorithm flexPrefTreeMatch. More type of flexibilities and preferences are being integrated in our project, to improve XML query languages.

References

[1] D. U. a. E. Z. Luigi Pontieri, "An approach for the extensional integration of data sources with heterogeneous representation formats," *Data & Knowledge Engineering,* vol. 45, pp. p291-331, 2003.

[2] S. B. S. C. V. D. A. A. F. F. G. e. a. Domenico Beneventano, "Semantic Integration and Query Optimization of Heterogeneous Data Sources," in *Advances in Object-Oriented Information Systems*, 2002.

[3] L. G. DeMichiel, "Resolving database incompatibility: an approach to performing relational operations over mismatched domains," Vols. 1, p 485-493, 1989.

[4] U. o. W. X. D. Repository, 14 March 2020. [Online]. Available: http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/www/repository.html#auctions.

[5] L. L. a. J. L. R. Rada CHIRKOVA, "Tractable XML data exchange via relations," *Frontiers of Computer Science,* pp. 243-263, 01 06 2012.

[6] D. C. L. L. a. M. F. Amano Shun'ichi, "On the tradeoff between mapping and querying power in XML data exchange," 09 2010.

[7] A. a. H. Ahmad, "XML-Based Data Exchange in the Heterogeneous Databases (XDEHD)," *International journal of Web & Semantic Technology,* vol. International journal of Web & Semantic Technology, pp. 11-24, 07 2015.

[8] K. S. a. M. Peter, "Integrating Unnormalised Semi-structured Data Sources," 2005.

[9] A. A. a. P. Jaroslav, "A Mediation Layer for Heterogeneous XML Schemas.," *IJWIS,* vol. 1, pp. 25-32.

[10] C. N. a. C. R. Tekli Joe, "Building Semantic Trees from XML Documents," *Journal of Web Semantics,* vol. 37, 03 2016.

[11] T. Mohammad, "Understanding Semantic Web and Ontologies: Theory and Applications," *Journal of Computing,* 06 2010.

[12] C.-B. N. C. F. a. R. O. Dernaika Farah, "Semantic Mediation for A Posteriori Log Analysis," in *ARES '19: Proceedings of the 14th International Conference on Availability, Reliability and Security*, 082019.

[13] H. M. a. D. Jérôme, "A Survey of XML Tree Patterns," *EEE Transactions on Knowledge and Data Engineering,* vol. 25, pp. 29-46, 01 2013.

[14] S. B. Sven Groppe, "Query Reformulation for the XML standards XPath, XQuery and XSLT," in *DBLP*, Berlin, January 2004.

[15] H. B. a. O. K. Saber Benharzallah, "Reformulating XQuery queries using GLAV mapping and complex unification," vol. Volume 28, no. 1, January 2016.

[16] P. Y. B. C. a. S. S. B. a. J. Xu, "AutoG: A Visual Query Autocompletion Framework forGraph Databases," Vols. 26, 347-372, 2017.

[17] Maurice Tchoupe Tchendji and Brice Nguefack, "Requêtes XPath bipolaires et évaluation," *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées, INRIA.,* 2017.

[18] T. W. L. Z. B. a. C. W. J. Lu, "Extended XML Tree Pattern Matching: Theories and Algorithms," in *IEEE Transactions on Knowledge and Data Engineering, vol. 23, no. 3*, March 2011.

[19] W, "Extensible Markup Language (XML) 1.0 (Fifth Edition)," [Online]. Available: https://www.w3.org/TR/xml/. [Accessed 12 08 2020].

[20] M. M. G. G. a. R. B. L. Ismael Sanz, "Fragment-based approximate retrieval in highly heterogeneous XML collections," *Data Knowl. Eng.,* vol. 64, pp. 266--293, 2008.

[21] W3C, "XML Path Language (XPath) 3.0," 08 04 2014. [Online]. [Accessed 05 04 2021].

[22] W3C, "XQuery 3.0: An XML Query Language," 08 04 2014. [Online]. Available: https://www.w3.org/TR/xquery-30/. [Accessed 04 03 2021].

[23] L. L. S. D. T. K. Jagadish H.V., "TAX: A Tree Algebra for XML," Berlin, Heidelberg, 2002.

[24] A. N. a. H. V. Jagadish, "Evaluating Structural Similarity in XML Documents," *WebDB,* pp. 61--66, 2002.

[25] H. V. J. L. V. S. L. a. S. P. Zhimin Chen, "From tree patterns to generalized tree patterns: on efficient evaluation of XQuery," in *In Proceedings of the 29th international conference on Very large data bases (VLDB '03)*, 2003.

[26] W. Y. L. L. V. S. a. J. H. V. Paparizos Stelios, "Tree Logical Classes for Efficient Evaluation of XQuery," 2004.

[27] H. T. a. H. M. S. Izadi Sayyed Kamyar, "S3: Evaluation of Tree-Pattern XML Queries Supported by Structural Summaries," vol. 68, p. 126–145, 2009.

[28] G. K. Werner Kießling, "Preference SQL — Design, Implementation, Experiences," Hong Kong, China, August 20-23, 2002.

[29] H. B. F. S. H. S. Kießling W., "Preference XPATH:A Query Language for E-Commerce," Heidelberg, 2001.

[30] S. C. a. M. Shiloach, "Flexible XML Querying Using Skyline Semantics," in *Proceedings of the 25th International Conference on Data Engineering, {ICDE} 2009, March 29 2009 - April 2 2009*, Shanghai, China, 2009.

[31] L. T. a. T. T. T. Maurice Tchoupé Tchendji, "A Tree Pattern Matching Algorithm for {XML} Queries with Structural Preferences," *CoRR,* vol. abs/1906.03053, 2019.

[32] M. T. T. a. P. J. Kenfack, "An XQuery Specification for Requests with Preferences on XML Databases," vol. 582, no. {IFIP} Advances in Information and Communication Technology, pp. 120--130, 2020.

[33] Y. T. a. G. F. a. a. B. S. Papadias, "Progressive skyline computation in database systems," *ACM Trans,* pp. 41–-82, 2005.

[34] S. a. K. D. a. S. K. Borzsony, "The Skyline Operator," in *Proceedings - International Conference on Data Engineering*, 2001.

[35] C. K. a. T. Tzouramanis, "A Survey of Skyline Query Processing," *ArXiv,* p. abs/1704.01788, 2017.

[36] K. B. P. I. a. V. Shanthi, "Goal Directed Relative Skyline Queries in Time Dependent Road Networks," *International Journal of Database Management Systems (IJDMS),* vol. 4, no. 2, pp. 01-12, 2012.

AUTHORS

Brice Nguefack received his master degree in computer science in 2016, from the department of mathematics and computer science, faculty of science, University of Dschang, Cameroon. Where he is actually a Ph.D. student. His research interests include Software Engineering, integration of preferences in XML query language, and XML preference query processing. (https://orcid.org/0000-0001-5241-2750)

Maurice Tchoupé Tchendji received his Ph.D degree in computer science in 2009, from the University of Yaounde 1, Cameroon and the University of Rennes 1, France. He is actually a senior lecturer in the department of mathematics and computer science, Faculty of Science, University of Dschang, Cameroun. His research interests include Distributed Computing, Theory of Computation, Software Engineering, integration of Preferences in XML query language, and XML preference query processing. (https://orcid.org/0000-0002-9208-6838)

Thomas Djotio Ndie received his Ph.D degree in computer science, in 2009, from National Advanced School of Engineering, University of Yaounde 1, Cameroon. He is actually an Associate Professor in the same University. He is also the team leader of the LIRIMA project name "Internet of Things for Developing countries". His research interests include Software Engineering, Wireless Mesh Network, Wireless Sensor Network, Intrusion Detection System, Open Wireless Access for Developing Countries. (https://orcid.org/0000-0002-6300-6237)