Optimizing XML Twig Queries with Full-Text Predicates

Ya-Hui Chang
Department of Computer Science and Engineering
National Taiwan Ocean University, Keelung, 202, Taiwan, R.O.C.
E-mail: yahui@ntou.edu.tw

ABSTRACT

Efficient query processing has been a critical issue for XML repositories. In this paper, we consider the XML query which can be represented as a query tree with twig patterns, and also consists of full-text constraints. Previously, the *structure-first* approach and the *keyword-first* approach have been proposed to process such kind of queries. The main focus of this paper is constructing an integrated system to support these two approaches and find the best execution plan. To achieve this goal, we first analyze the components of these two approaches and design a set of operators. We then derive the corresponding cost model and rewriting rules to perform cost-based optimization. We also propose several heuristic rules by observing the behaviors of the two approaches. Via an extensive experimental study, we demonstrate that our cost-based system and heuristic system are both effective.

1. INTRODUCTION

As the XML (eXtensible Markup Language) technology emerged as the de facto standard for information sharing and data exchange on the Web, XML data management and query processing have attracted a lot of attention from the academic and business communities.

In general, the nested structure of an XML document is captured by a tree model, so XML queries, *e.g.*, XPath or XQuery, are normally specified based on path expressions to navigate the complex structure of XML data. The path expressions involved in a query might constitute a tree structure, and such query is usually named as a *twig query*. There have been many research efforts on the relevant processing techniques [2, 3, 4, 9]. They usually first identify the elements which meet the tag or path requirements, and apply specially-designed encoding schemes, algorithms, or data structures, *e.g.*, indices or stacks, to expedite the combining process.

In contrast, researchers also advocate keyword-based search against XML documents, since it provides a more friendly user environment. Specifically, users do not need to explicitly state the structural constraint, but the system will ensure that the returning data satisfy the structure of the queried XML document [1, 7, 8, 11]. This type of researches usually apply the techniques seen in the information retrieval (IR) field, *e.g.*, inverted lists or ranking schemes.

In view of the need to combine the above two querying facilities, XQuery and XPath Full-Text (XQFT) 1.0, has been proposed and became a W3C standard [10]. Consider the following sample query, which is posed against the XML document in Figure 1:

Q1
for $p in /catalog/item
where $p/remark **contains text** (“database” ftand “design” **ordered**) ftand (“database” ftand “design” **distance** at least 2 words) and $p//name **contains text** (“Peter” ftand “Rob” **ordered**)
return $p

In this query, the path expressions “*/catalog/item*”, “*/catalog/item/remark*” and “*/catalog/item//name*” form a twig pattern, which represent the *structural constraint* on the retrieved data. On the other hand, the operator *“contains”* is used to enforce the content of the elements *remark* and *name* to satisfy the given *full-text constraints*. Particularly, the content of the element “/catalog/item/remark” is required to have an ordering between the keywords “database” and “design”, with the distance at least “2” between them.

We have previously proposed two ways to process XML queries with these constraints [3]. The first one is called the *structure-first* approach. It mainly follows the research direction originally designed for processing twig queries, but is extended to be able to process full-text constraints as well. In contrast, the second one is called the *keyword-first* approach, since it is mainly based on the techniques for processing keyword-based queries. This approach will first process the full-text constraints, and then make its answer also satisfy the structural constraint. Although these two approaches have been shown both feasible, they were built as two separate systems where users are hard to determine which one to use.

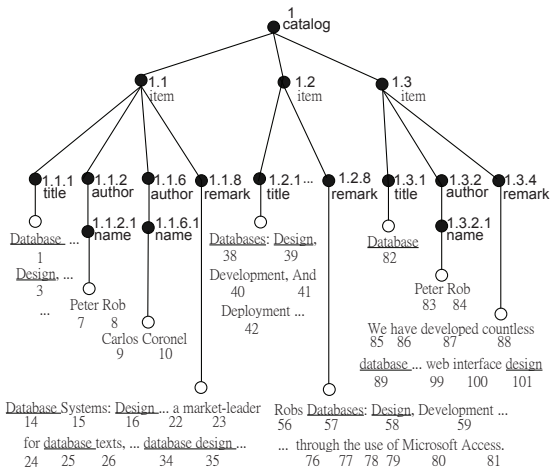


Figure 1: The Sample XML Tree

In this paper, we intend to construct an integrated system to support the *structure-first* (SF) approach and the *keyword-first* (KF) approach in a uniform way. We first analyze the components of these two approaches and design a set of operators to represent how they work. We then define the cost model to represent the cost of each operator and propose a set of rewriting rules, so that we can choose the plan with the least cost from all possible execution plans. In addition to supporting cost-based optimization, we also propose several heuristic rules by observing the behaviors of the two approaches. Finally, we have conducted a series of empirical studies. The experimental results show the effectiveness of the cost-based optimization system and the proposed heuristic rules.

The remaining of this paper is organized as follows. We first introduce the underlying data model and the two approaches in Section 2. We then describe how we achieve cost-based optimization in Section 3. We have performed a series of experiments to demonstrate the effectiveness of our integrated system. The experimental results are analyzed in Section 4. Finally, conclusion and future works are discussed in Section 5.

2. PRELIMINARIES

In this section, we first discuss the underlying data representation in our system. We then explain the SF approach and the KF approach. The XML tree in Figure 1 and the sample query $Q1$ will be used as the running example throughout this paper.

2.1 Data Modeling

The XML document is represented as a rooted labeled tree as usual. To quickly determine the structural relationship between two elements, each element node is associated with the extended Dewey encoding [9]. There are mainly two reasons for applying this encoding scheme. First, this encoding scheme has the main properties of the original Dewey encoding, so we can easily obtain the LCA (lowest common ancestor) of two elements by computing their common prefixes. This is required by the keyword-first approach. For example, the LCA of elements 1.1.1 and 1.1.2 is 1.1. Second, a unique property of the extended Dewey encoding is that it can be easily transformed to a labeled path. For example, encoding 1.1.1 can be transformed to the path "/catalog/item/title". This supports the efficient processing of twig joins.

For processing full-text constraints, each keyword in the context is also given a *global* position, which is assigned across elements. For example, the first "Database" under the leftmost *title* element is encoded as 1, while the first "Database" under the leftmost *remark* element is encoded as 14.

2.2 The Structure-First Approach

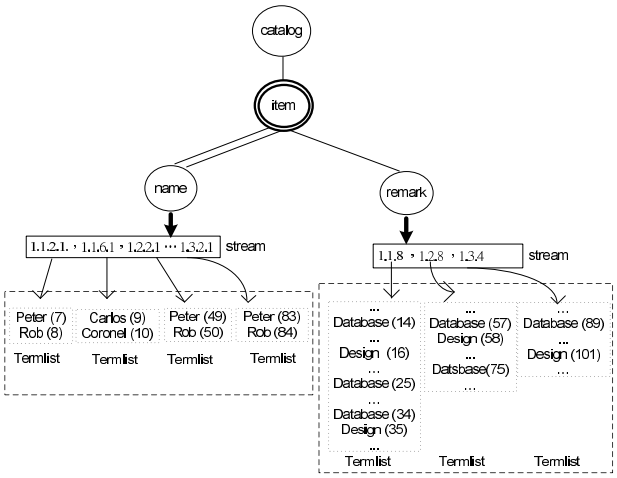


Figure 2: The Query Tree with Streams and Termlists

In the SF approach, an input query is first constructed as a query tree to clearly show its structural constraint. The query tree is built based on all the path expressions specified in the query, where the component elements are illustrated by nodes and the location steps are denoted by edges. The node with a double circle indicates the answer node. After the query tree is built, the SF approach first identifies the elements which match the tag constraint for each leaf node of the query tree and represents them in a sorted order, which are called the *stream* of the associated node. For each element in the stream, the SF approach then represents the component keywords along with their positions in a sorted order, which are named as the *Termlists*. Figure 2 illustrates the result of processing *Q1* against the sample XML tree at this stage.

The SF approach then examines each element in the stream, and uses a variant of the merge-sort algorithm, to see if the associated Termlist represents the queried keywords in the required *ordered* or *distance* sequence. For example, in the stream of the node *name*, the elements 1.1.2.1, 1.2.2.1, and 1.3.2.1 all satisfy the *ordered* constraint for the keywords *Peter* and *Rob*. After identifying all the elements which satisfy the full-text constraint from each individual stream, the SF approach applies a twig join algorithm, called TJFast [9], to find the correct answers which satisfy the whole twig constraint.

2.3 The Keyword-First Approach

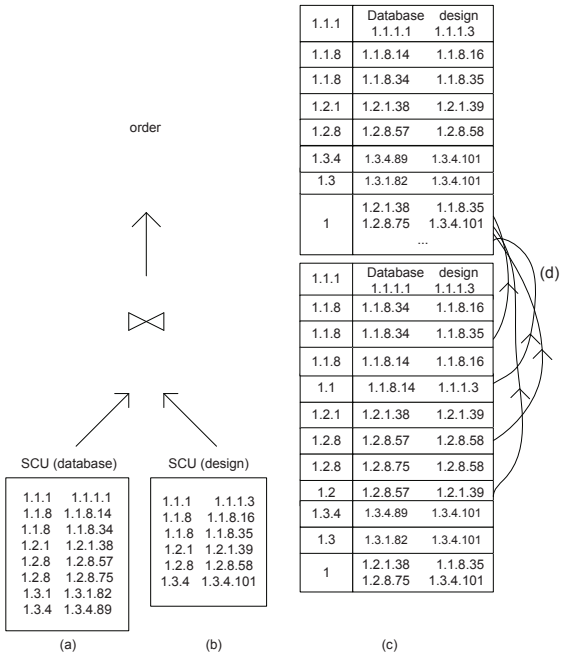


Figure 3: An Example of the KF Approach

In contrast to Termlists, the underlying data structure of the KF approach is the SCU (Smallest Containing Unit) [1]. An SCU consists of a list of items, where each item represents the element and the position which matches the pattern (keyword or full-text constraint). For example, in Figure 3(a)-(b), we can see the SCU tables for the keywords "database" and "design", where the position is represented by attaching the global position to the extended Dewey encoding of its parent element. Another difference from the SF approach is that the items in an SCU table are sorted based on the postorder of nodes, instead of preorder.

The KF approach starts by processing the full-text constraints. For each full-text constraint which consists of the keywords K1 and K2, it will first create the corresponding SCU tables (Figure 3(a)-(b)). It then calculates the LCAs which represent both keywords K1 and K2, as seen in Figure 3(c). To identify those elements which satisfy the full-text constraint, the algorithm first determines if an LCA satisfies the given constraint. If it does not, the matched position will be sent to its nearest ancestor to check next. For example, as shown in Figure 3(d), element 1.1 does not satisfy the first ordered constraint in $Q1$, because the matched position for "database" (1.1.8.14) is bigger than the matched position for "design" (1.1.1.3). Therefore, these information will be propagated to its ancestor (element 1) for further checking. Note that in this approach, it is comparably easy to check the full-text predicates *ordered* and *distance*, since we can directly retrieve the matched positions for the keywords from the SCU, and a simple arithmetic calculation will suffice. After obtaining those elements which satisfy the full-text constraints, the KF approach will represent them as streams of the corresponding leaf node in the query tree, and find answers by invoking the TJFast algorithm.

3. COST-BASED OPTIMIZATION

In this section, we discuss how we combine the SF and KF approaches into an integrated system, and how to perform cost-based optimization.

3.1 Operators

Table 1: Operators

operator	cost model
T(t)	$c_1 * \|T(t)\|$
K(k)	$c_2 * \|K(k)\|$
$C_{op}^{t1,t2}$(.)	$c_3 * \|I\|$; I: input Termlist
	$c_4 * \|I\|$; I: input SCU
P_p(.)	$c_5 * \|I\|$; I: input Termlist
TJ_{qt}(stream*)	$c_6 * \|input\| + c_7 * \|output\|$
L(s1, s2)	$c_8 * (\|s1\| + \|s2\|) + c_9 * \|output\|$
O(s)	$c_{10} * \|s\|$

By analyzing all components of the SF and KF approaches, we define a set of operators for the integrated system, as summarized in Table 1. Among the seven operators, the first two operators are classified as I/O operators, the following two operators are filters, and the last two operators are introduced because they are required during the process of the KF approach. Specifically, operators T and K identify elements based on the given tag t and keyword k, respectively. Operators C and P return elements satisfying the full-text constraint based on op, $t1$ and $t2$, and the path constraint p, respectively. Operator TJ identifies a set of *match trees* from the given streams. A match tree is a set of elements, where each component satisfies individual constraints, and the whole set satisfies the structural constraint imposed by the twig query. Finally, the *L* operator is applied to identify the LCAs which consist of the input keywords. The

O operator is required since the elements represented in an SCU table are in postorder, while the elements represented in a stream should be in preorder. This operator will perform the required order transformation.

We can apply these operators to represent different execution plans. The following expression describes a possible execution plan for *Q1* based on the SF approach, where the keywords and tag names are represented in shorthand, and qt represents the sample query tree depicted in Figure 2:

$$TJ_{qt}(P_{/c/i/r}(C^{dat,des}_{dis>=2}(C^{dat,des}_{ord}(T(remark)))),\\ P_{/c/i//n}(C^{Pet,Rob}_{ord}(T(name)))) \quad (1)$$

Specifically, this plan first processes the leaf nodes of the query tree and identifies those elements representing the given tag names. It then examines each element in the associated stream and determines if the corresponding Termlist satisfies the specified full-text predicates. Finally, it picks those elements satisfying the path constraints to form match trees and return answers.

The following expression represents another possible execution plan based on the KF approach:

$$TJ_{qt}(O(P_{/c/i/r}(C^{dat,des}_{dis>=2}(C^{dat,des}_{ord}(L(K(dat),K(des)))))),\\ O(P_{/c/i//n}(C^{Pet,Rob}_{ord}(L(K(Pet),K(Rob)))))) \quad (2)$$

Note that the *L* and *O* operators are additionally required in the KF approach.

3.2 Rewriting Rules

Table 2: Rewriting Rules

No	Rule
1	$C^{t1,t2}_{op1}(C^{t3,t4}_{op2}(.)) = C^{t3,t4}_{op2}(C^{t1,t2}_{op1}(.))$
2	$P_p(C^{t1,t2}_{op}(.)) = C^{t1,t2}_{op}(P_p(.))$
3-1	$C^{t1,t2}_{op1}(C^{t3,t4}_{op2}(L(L(s1,s2),(L(s3,s4)))) =$ $C^{t1,t2}_{op1}(C^{t3,t4}_{op2}(L(L(L(s1,s2),s3),s4))) =$
3-2	$C^{t3,t4}_{op2}(L(C^{t1,t2}_{op1}(L(s1,s2)),(L(s3,s4)))) =$
3-3	$L(C^{t1,t2}_{op1}(L(s1,s2)),C^{t3,t4}_{op2}(L(s3,s4))) =$

Table 2 lists several rewriting rules designed for this system. Rule 1 indicates that two C operators are commutable; rule 2 indicates that a C operator and a P operator are commutable. These two rules are applicable to both SF and KF approaches. There are certain rewriting rules which are only applicable in one approach. For example, rules 3-1 to 3-3 represent that the C operator can push before or after the LCA operation, which only function in the KF approach.

We then explain how to produce different execution plans. First, for each leaf node of the query tree, we derive the default-SF plan and the default-KF plan. The default-SF plan is produced by enforcing the following processing sequence: the ordered constraint, the distance constraint, and the path constraint. The default-KF plan performs the LCA operation first, and then follows the same sequence as in the default-SF plan.

After getting the default plans, as the sample execution plans (1)-(2) presented in Section 3.1, we then apply the rewriting rules to produce all possible plans. For example, rewriting rule 2 can be applied to execution plan (1) and make path filtering performed before the full-text constraint. Finally, we consider possible combinations among all leaf nodes to produce the set of complete execution plans.

3.3 The Cost Model

Table 3: Coefficients in the Cost Model

coefficient	c_1	c_2	c_3	c_4	c_5
value	12	15	1.56	1.33	4.24
coefficient	c_6	c_7	c_8	c_9	c_{10}
value	1	4	1	5	0.1

In our cost-based optimization system, we estimate the cost of each possible execution plan and choose the plan with the least cost. To do so, we design the cost model for each operator, as shown in the last column of Table 1. They are derived based on the time complexity of the corresponding algorithms, which are basically linear to the input (and/or output).[1]

The coefficients in the cost model, *i.e.*, c_1 through c_{10}, are obtained empirically based on representative datasets and queries. Briefly speaking, we divide the execution time of a query into several portions based on the operators which constitute the execution plan, and also record the amounts of data operated (and produced) within each portion. By simple calculation we get the values of coefficients. We execute each query twenty times and get the average values of coefficients. The values got from all test queries are again averaged and normalized. The final result is summarized in Table 3. In addition, we also collect several types of statistic data, which include the occurrences of each tag name and the occurrences of each keyword in an XML document, to calculate the estimated cost of an execution plan.

4. EXPERIMENT

We have designed several experiments to evaluate the performance of our integrated system. All the experiments are performed on a personal computer with Intel Core i7 3.7GHz CPU and 16GB memory, with the Microsoft Windows 7 operating system. The test queries are summarized in Table 4, where the XPath-like syntax is applied to save space. Besides, "cnt" is the shorthand

[1]We build two indices to support the two I/O operators. The first one is a hash index designed for operator T, which has the tag-name as the key, and will return the information: *(encoding, keyword, position)*. The second one is an inverted index designed for operator K, which utilizes the *keyword* as the keys, and returns the *encoding* and *position* information.

Table 4: The Test Queries

No	Query Statement
Q1	/dblp/inproceedings[./title cnt (“system”, “system”), (“advanced”, “course”), (“algorithm, “application”), (“base”, “computer”), (“information”, “image”) and ./booktitle cnt (“language”, “language”), (“conference”, “conference”), (“data”, “performance”), (“model”, “management”), (“design”, “design”)]
Q2-5	*similar to Q1 and omitted due to space limitation*
Q6	/dblp/inproceedings[./booktitle cnt (“air”, “air”) and ./title cnt (“consider”, “consider”)]
Q7	/dblp/inproceedings[./booktitle cnt (“architecture”, “architecture”) and ./title cnt (“knowledge”, “knowledge”)]
Q8	/dblp/inproceedings[./booktitle cnt (“system”, “system”) and ./title cnt (“us”, “us”)]
Q9	/dblp/inproceedings[./school cnt (“university”, “university”) and ./note cnt (“conference”, “conference”)]
Q10	/dblp/inproceedings[./editor cnt (“John”, “John”) and ./publisher cnt (“Germany”, “Germany”)]
Q11	/dblp/inproceedings[./title cnt (“System”, “System”) and ./author cnt (“Jerry”, “Jerry”)]
Q12	/site/reqions/asia/item[.//description//text//keyword cnt (“master”, “attend”, <=, 500) and .//shipping cnt (“ship”, “see”)]
Q13	/site/reqions/asia/item[.//description//text cnt (“master”, “attend”, <=, 500) and .//shipping cnt (“ship”, “see”)]
Q14	/site/reqions/asia/item[.//description cnt (“master”, “attend”, <=, 500) and .//shipping cnt (“ship”, “see”)]
Q15	/site/reqions/asia/item[.//desciption//text cnt (“master”, “attend”, <=, 5) and .//shipping cnt (“see”, “see”)]
Q16	/site/reqions/asia/item[.//desciption//text cnt (“master”, “attend”, <=, 50000) and .//shipping cnt (“see”, “see”)]
Q17	/site/reqions/asia/item[.//desciption//text cnt (“master”, “attend”, <=, 50000000000) and .//shipping cnt (“see”, “see”)]
Q18	/site//item[.//desciption//text cnt (“master”, “master”) and .//keyword cnt (“bound”, “bound”) and .//shipping cnt (“description”, “description”)]
Q19	/site//item[.//desciption//text cnt (“master”, “master”) and .//keyword cnt (“bound”, “bound”) and .//shipping cnt (“description”, “description”) .//location cnt (“Netherland”, “Netherland”)]
Q20	/site//item[.//desciption//text cnt (“master”, “master”) and .//keyword cnt (“bound”, “bound”) and .//shipping cnt (“description”, “description”) .//location cnt (“Netherland”, “Netherland”) and .//from cnt (“june”, “june”)]

of *contains text*, (A, B) represents “A ftand B ordered”,[2] and the distance constraint is represented as a quadruple. These test queries are designed to consist of a variety of cases. Q1-Q5 have the same query tree structure, but with 10, 14, 18, 22, 26 full-text constrains, respectively. Q6-Q8 also have the same query tree structure, but the constrained keywords have different frequencies, where those in Q6 have the lowest frequencies, and those in Q8 have the highest frequencies. In contrast, Q9-Q11 have different tag frequencies, where those in Q9 have the lowest frequencies, and those in Q11 have the highest frequencies. Q12-Q14 have the same full-text constraint, but the length of the root-to-leaf path decreases. The difference among Q15-Q17 is that the distance between the constrained keyword increases. Finally, Q18-Q20 have increasing numbers of path constraints.

We apply three datasets, which are two DBLP collections with the size 107MB and the size 872MB, respectively, and the XMark data set with the size 116MB. Each query is executed three times, and we calculate the average of the last two execution time.

[2]When using the same word in an ordered constraint, *e.g.*, (“system”, “system”), we only require the word to appear once.

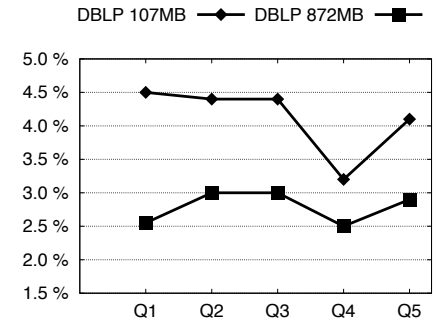


Figure 4: Overhead of Performing Optimization

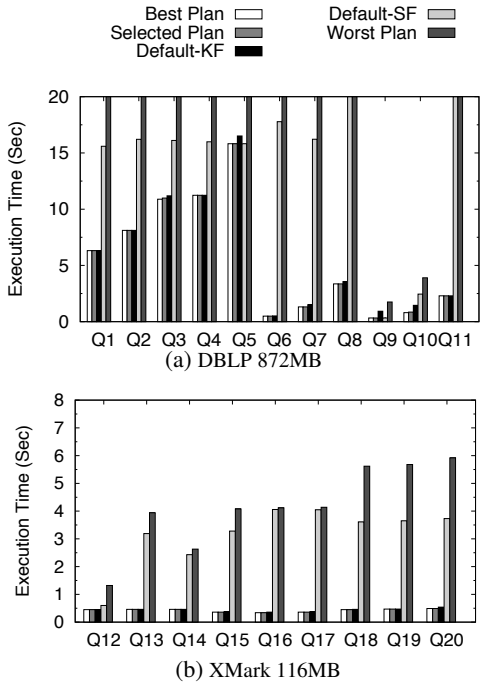


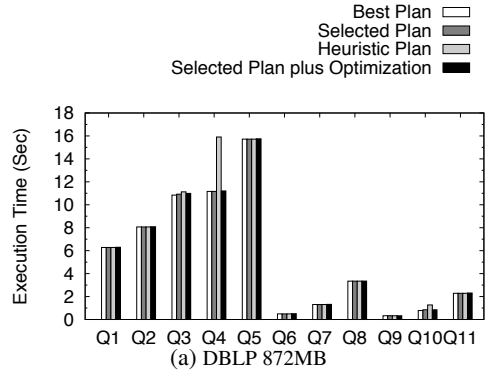
Figure 5: Effectiveness of Cost-Based Optimization

4.1 Effectiveness of Cost-based Optimization

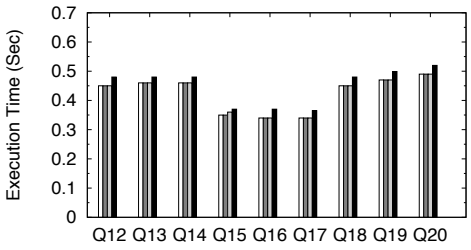
First, we evaluate the overhead of performing optimization. Normally, when a query consists of more constraints, more alternative plans will be produced and need to be explored. However, the query might also need more time to evaluate. In Figure 4, we show the ratios of the optimization time and the execution time of selected plans. Observe that they are all below 4.5%, which are quite minor and acceptable.

Next, we evaluate the performance of our cost-based optimization system.[3] For each query, we record the execution time of the real best plan, the plan selected by our system, the default-KF plan, the default-SF plan, and the worst plan. We applied queries Q1-Q11 to the two DBLP datasets and applied queries Q12-Q20 to the

[3]Experimental results of the smaller DBLP dataset are similar to those of the larger one, and are omitted in Figures 5-6.



(a) DBLP 872MB



(b) XMark 116MB

Figure 6: Effectiveness of the Heuristic Rules

XMark data set. As shown in Figure 5, for the vast majority (over 90%) of queries, our cost-based system will choose the best plan. If not, the selected plan whose execution time is all less than 8% above the cost of the actual best plan. This shows the effectiveness of our system.

Based on the execution time of each query, we observe that the KF approach usually outperforms the SF approach. The reason is that the costs of those operators comprising the KF approach and SF approach do not differ a lot. The only exception is that coefficients $c1$ and $c2$, *i.e.*, the costs of the input operations, are larger than others by an order of magnitude (See Table 3). Therefore, the amount of the input dominates the total cost. Since a document usually consists of more keywords than tags, keyword frequencies tend to be less than the tag frequencies. Therefore, it is reasonable that the KF approach is commonly more efficient than the SF approach when the query is not very complex. However, recall that the KF approach needs to do extra LCA computation and order transformation. Therefore, we can see that the KF approach will be defeated when there are many full-text constraints, as shown in Q5.

4.2 Effectiveness of the Heuristic Rules

Based on the above discussion and observing that the best plan is usually the default plan (Figure 5), we derived two heuristic rules as listed below:

1. If sum(Km) $<$ sum (Tm), choose the default-KF plan.
2. If the number of full-text constraints (N) is bigger than 20, choose the default-SF plan.

In short, our heuristic system will not apply the rewriting rules discussed in Section 3.2, but directly choose the most appropriate plan based on the following rule:

If (sum(KM) $<$ sum(Tm)) or $N < 20$ **then** choose the default-KF plan; **else** choose the default-SF plan.

As shown in Figure 6, our heuristic system works correctly in about 80% cases. When additionally considering the optimization time required by the cost-based system, our heuristic system runs faster than the cost-based system in more than 90% cases. This shows the effectiveness of our heuristic rules.

5. CONCLUSION

For an XQuery with structural and complex full-text constraints, we first design a cost-based optimizer and then derive a heuristic system to avoid performing rewriting. The experimental results show that the two approaches are both effective. In the future, we plan to explore more optimization techniques such as described in [5, 6] to further improve querying performance.

6. REFERENCES

[1] S. Amer-Yahia, E. Curtmola, and A. Deutsch. Flexible and efficient xml search with complex full-text predicates. In *Proceedings of the SIGMOD Conference*, 2006.

[2] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal xml pattern matching. In *Proceedings of the ACM SIGMOD conference*, 2002.

[3] Y.-H. Chang, C.-Y. Wu, and C.-C. Lo. Processing xml queries with structural and full-text constraints. *Journal of Information Science and Engineering*, 28(2), 2012.

[4] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K. S. Candan. Twig2stack: Bottom-up processing of generalized-tree-pattern queries over xml documents. In *Proceedings of the 32rd VLDB Conference*, 2006.

[5] H. Georgiadis, M. Charalambides, and V. Vassalos. Cost based plan selection for xpath. In *Proceedings of the SIGMOD conference*, 2009.

[6] H. Georgiadis, M. Charalambides, and V. Vassalos. Efficient physical operators for cost-based xpath execution. In *Proceedings of the EDBT conference*, 2010.

[7] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In *Proceedings of the SIGMOD Conference*, 2004.

[8] Z. Liu and Y. Chen. Reasoning and identifying relevant matches for xml keyword search. In *Proceedings of the 34rd VLDB Conference*, 2008.

[9] J. Lu, T. W. Ling, C.-Y. Chan, and T. Chen. From region encoding to extended dewey: On efficient processing of xml twig pattern matching. In *Proceedings of the VLDB conference*, 2005.

[10] The World Wide Web Consortium. XQuery and XPath full text 1.0. w3c recommendation. http://www.w3.org/TR/xpath-full-text-10/, 2011.

[11] M. Theobald, R. Schenkel, and G. Weikum. An efficient and versatile query engine for topx search. In *Proceedings of the VLDB Conference*, 2005.