XBLEND: Visual XML Query Formulation Meets Query Processing

Zhou Yong   Sourav S. Bhowmick   Erwin Leonardi   K. G. Widjanarko

Singapore-MIT Alliance, Nanyang Technological University, Singapore
School of Computer Engineering, Nanyang Technological University, Singapore
{zhouyong, assourav, lerwin, klarinda}@ntu.edu.sg

***Abstract*— Due to the complexity of XML query languages, the need for visual query interfaces that can reduce the burden of query formulation is fundamental to the spreading of XML to wider community. We present a RDBMS-based XML query evaluation system, called XBLEND, that takes a novel and non-traditional approach to improving query performance by *blending* visual query formulation and query processing. It exploits the latency offered by GUI-based visual query formulation to prefetch portions of the query results. The basic idea is that we prefetch constituent path expressions, store the *synopsis* of intermediary results, reuse them when connective is added or "Run" is pressed. In our demonstration we show that our system exhibits promising performance in evaluating XML queries and show its usefulness in life sciences domain.**

I. Introduction

XML has emerged as the leading textual language for representing and exchanging data over the Web in a wide variety of domains. For instance, increasingly, data in life sciences are being represented in XML format (e.g., SBML, MAGE-ML). The aftermath of this growing acceptance of XML by a wide spectrum of applications is that they are generating huge volumes of data as well as "consumers" (users). This has generated tremendous interest in the mainstream database community to propose innovative solutions for storage and query processing of large volumes of XML data.

Querying XML data involves two key steps: *query formulation* and *efficient processing* of the formulated query. An XML query language, such as XPath or XQuery, can be used to formulate a query in textual form. Unfortunately, formulating such query often demands considerable cognitive effort from the end user (consumer) and requires "programming" skills that is at least comparable to SQL. However, in many real life applications it is not realistic to assume that users are proficient in expressing such textual queries. This issue has been aptly summarized by Abiteboul et al. [1]:

> *"Thirty years of research on query languages can be summarized by: we have moved from* SQL *to XQuery. At best we have moved from one declarative language to a second declarative language with roughly the same level of expressiveness. It has been well documented that end users will not learn* SQL*; rather* SQL *is notation for professional programmers."*

Hence, the need for easy and intuitive techniques that can reduce the burden of query formulation is fundamental to the spreading of XML to wider community. Consequently, there has been long stream of research in visual query languages in the context of XML databases [3]. Majority of these efforts have focused on providing different degree of query expressiveness claiming that the proposed interfaces are user-friendly and speed up query construction.

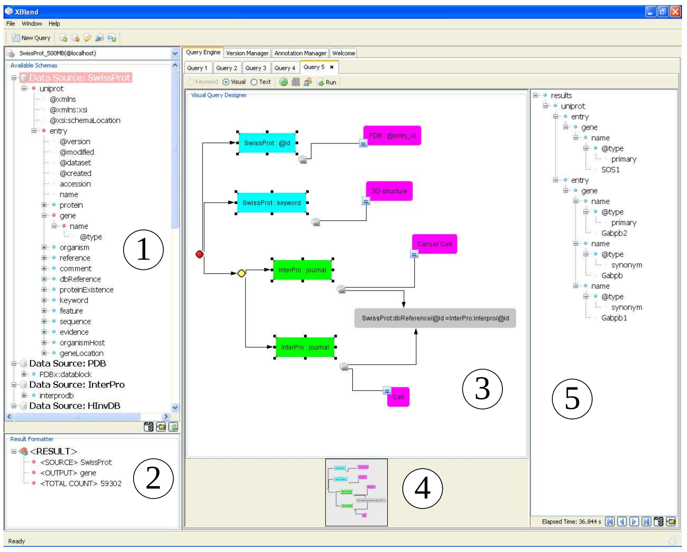
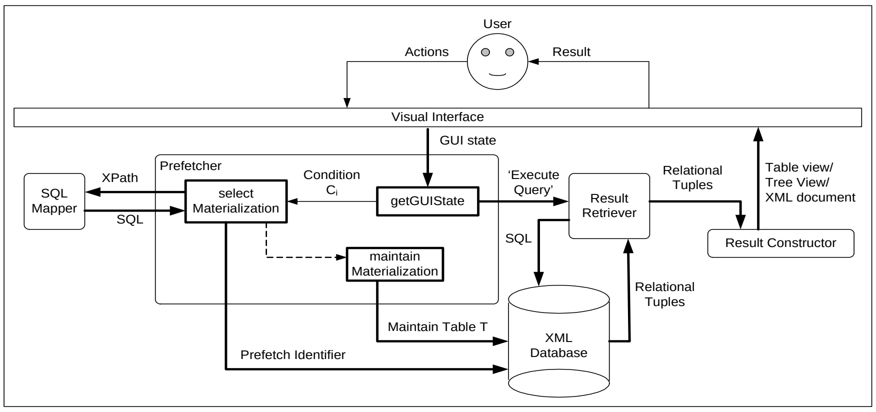


Fig. 1. Visual interface of XBLEND.

The database community has invested significant effort in devising innovative and powerful storage structures and query optimization mechanisms on top of relational as well as native framework to support efficient evaluation of XML queries [5]. Typically, these efforts are independent of any query formulation strategy. In particular, the relational approach has gained popularity due to its stability, efficiency, expressiveness, and its wide spread usage in the commercial world. In spite of significant progress in efficient processing of XML queries in relational environment, as we shall see in Section 4, fast evaluation of *complex* XML queries involving joins and AND/OR connectives is still a challenging problem.

This demonstration features a novel XML query evaluation system, called XBLEND, that for the first time ***blends** the two orthogonal areas of visual* XML *query formulation and query processing in order to improve query performance*. We take a novel and non-traditional approach to improving query performance by exploiting the latency offered by the GUI-based query formulation to *prefetch portions of the query results in a single-user environment* [2], [7]. The key advantages of



(a) System Architecture

```
Input: Actions from the query interface.
Output: Table T.

Initialize T = {}
State S = getGUIState()
while (S != "Execute Query") or
    (prefetching task list is not empty) do
    /* Materialize identity paths */
    C_i = Next condition in task list
    M_i = selectMaterialization(C_i)
    T = maintainMaterialization(T, M_i)
    S1 = getGUIState()
    /* wait till GUI state changes */
    while S1 == S and prefetching task list empty do
      S1 = getGUIState()
    end while
end while
```

(b) Prefetching Algorithm

1. Specify '/catalog/book' as the result subtree
2. Specify C1: **(retail_price > 10 and retail_price < 30)**
3. Specify C2: **(retail_price > 40 and retail_price < 50)**
4. Change operator connecting C1 and C2 to OR
5. Specify C3: **publish_year = '2007'**
6. Specify C4: **retail_price < recommend_price]**
7. Click 'Run' button

(c) Sequence of Actions

Fig. 2. System architecture of XBLEND and prefetching process.

XBLEND are as follows. Firstly, our optimization technique is built *outside* the relational optimizer and is orthogonal to any other existing optimization techniques. Secondly, our approach is not restricted by the underlying schema of the database. As a result, it can easily be integrated with any relational storage approaches. Thirdly, the prefetching-based query processing is transparent from the user.

II. MOTIVATING EXAMPLE

Querying biological data across multiple sources is a key activity for many biologists. If these sources represent data in XML format (e.g., INTERPRO (www.ebi.ac.uk/interpro/), SWISSPROT (www.expasy.ch/sprot/), PDB (www.pdb.org)) then biologists need to be familiar with XML query languages to be able to formulate meaningful queries over these data sources. For instance, suppose a biologist has retrieved XML representation of INTERPRO, SWISSPROT, and PDB, and stored them using XML support provided by MSSQL Server 2005. He would now like to *find the list of genes from the* SWISSPROT *entries that have the keyword "3D-structure" and have a publication in the journal "Cell" or "Cancer Cell" and has a database reference to the cell category of* PDB *database*. This query requires a join between the three downloaded data sources. The textual form of the query is shown in Figure 6 (Query $Q6$). However, formulation of this textual query by a biologist can be both time-consuming and error-prone. Importantly, it is not reasonable to expect him to learn the complex syntax of XQuery. Hence to ease query formulation, a visual interface (such as Figure 1) can be built on top of MSSQL Server.

Although query formulation now becomes significantly easier, evaluation of "multiple-sources" queries is a challenging problem as it may not be supported by relational-based XML database systems or it is expensive to evaluate. In fact, in this case, MSSQL Server is not able to evaluate this query as it does not support join across different data sources. On the other hand, IBM DB2 $v9.5$ takes more than 5 minutes to evaluate this query. In this demonstration, we present XBLEND that can return results of the above query in *less than 35s* by exploiting the latency offered by visual query formulation.

III. SYSTEM OVERVIEW

Figure 2(a) shows the system architecture of XBLEND and consists of the following modules. We use our own RDBMS-based XML query processor [8] as the underlying database.

Visual Interface Module: Figure 1 depicts the screen dump of the current version of the visual interface. It consists of five panels. The left panel (Panel 1) displays structure/schema of different XML data sources that are stored in our system. The user chooses one or more (in case of join) of the target sources over which queries are formulated. To formulate a query, the user first selects the nodes (subtrees) in Panel 1 that should be present in the query results and drags it to Panel 2. For instance, in Figure 1, the subtree selected is `gene` indicating that the user only wants to view information related to genes in the result. The *Visual Query Designer* panel (Panel 3) depicts the area for formulating XPath queries. The user drags the node to be queried from Panel 1 and drops it in this panel. A *Condition Dialog* appears and the user is expected to fill in the condition (if any) that should be satisfied by the selected node. The user can combine two or more conditions using AND/OR (default is AND) connectives. To provide more user-friendly navigation in Panel 3, a satellite view (Panel 4) is provided with zooming functionality. The user can execute the query by clicking on the "Run" icon in the *Query Toolbar*. The *Results View* (Panel 5) displays the query results.

Prefetcher Module: This module implements the prefetching algorithm (Figure 2(b)) [2], [7]. In brief, the algorithm works as follows. Each action made by the user through GUI is monitored by the `getGUIState` function. Once the user specifies the first condition, the prefetching process begins and will terminate when all prefetching steps are completed. Suppose a user formulates a condition C (e.g., related to genes in Figure 1) in Panel 3. The `selectMaterialization` function retrieves and materializes the *identifiers* of the `gene` subtrees in the XML document(s) that satisfy C. It returns a table M that contains these *identifiers* only. In our system, we use the *DeweyOrderSum* [8] of the left-most descendant leaf node of a `gene` subtree as an identifier. Note that the subtree identifier is not tightly coupled to any specific system as any numbering scheme that can uniquely identify subtrees

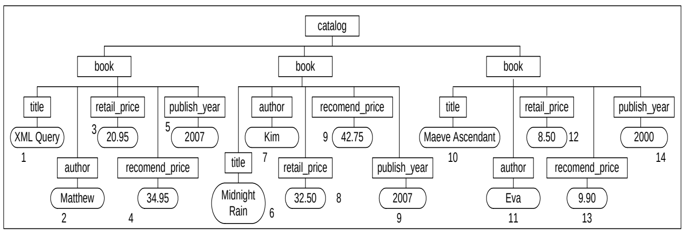


Fig. 3. Example of XML document.

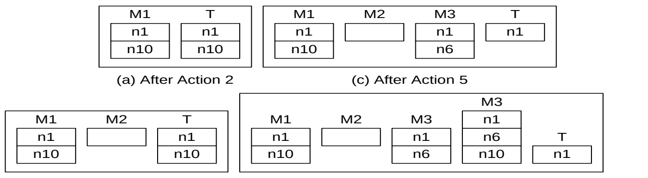


Fig. 4. The M_i and T tables during prefetching.

Source	Size	No. of Files	No. of Attributes	No. of Leaf Elements	No. of Internal Elements	Depth
XBench	1 GB	1	1,500,000	6,760,348	15,682,264	8
SwissProt	500 MB	1	13,606,352	7,795,349	2,373,031	6
PDB	287 MB	30	22,980	20,670	175,264	4
InterPro	50 MB	1	944,564	268,714	485,890	5

Fig. 5. Datasets.

in an XML tree can be used as an identifier. For instance, the *preorder* and *dewey order* values of nodes can be used for *region encoding* and *dewey number-based* labeling schemes, respectively [5]. Also, note that we materialize the identifiers instead of the entire subtrees because it is more efficient in terms of execution time and storage requirement (the size of materialized identifier table is always smaller than or equal to the table containing entire materialized subtrees). Finally, the `selectMaterialization` function also determines if C will return any results and notify users accordingly.

Next, the algorithm invokes `maintain Materialization` function. Since each prefetching operation is useful for the next, this function helps to replace existing materializations with new materializations preferably using the previous materializations. It maintains the set of identifiers (in table T) that satisfy the query conditions formulated by the user at a given timepoint. Finally, after the user clicks "Run" button, the *Prefetcher* module passes the table T containing the identifiers of final matching subtrees to the *Result Retriever* module.

Let us now illustrate the *Prefetcher* module with an example. Consider the XML document in Figure 3 and the XPath query */catalog/book[((retail_price>10 and retail_price < 30) or (retail_price >40 and retail_price <50)) and publish_year='2007' and retail_price < recommend_price]*. The sequence of GUI actions undertaken by a user to formulate this query is shown in Figure 2(c). Particularly, the condition C_1 will initiate the prefetching process. Our system retrieves all the identifiers of `book` subtrees that satisfy C_1 and stores them in the table M_1 (Figure 4(a)[1]). Then, the table T (initially empty) will be maintained by `maintainMaterialization` function as shown in Figure 4(a). Next, the user specifies the condition C_2. The *Prefetcher* retrieves the identifiers of `book` subtrees that satisfy C_2, stores them in M_2, and maintains table T. M_2 and T are depicted in Figure 4(b). Observe that M_2 is empty (no subtree satisfies C_2). Consequently, T remains unchanged. At this point of time, our system realizes that C_2 is not going to contribute to the final results and as a result action 4 specified by the user will not be processed. Similarly, M_3 in Figure 4(c) results from condition C_3 and $T = M_1 \cap M_3$. Next, the last condition C_4 is formulated and M_4 is materialized (Figure 4(d)) by the *Prefetcher*. The `maintainMaterialization` function determines that $M_4 \supset M_1$ and $M_4 \supset M_3$. Consequently, it will not update table T. Finally, the user clicks on "Run" icon and the *Prefetcher* module sends T containing the identifier of the first `book` subtree to the *Result Retriever* module.

[1]Note that the identifier of the left-most descendant leaf node (denoted as n_i) of a `book` subtree is used as the identifier of the subtree.

SQL Mapper Module: This module translates the XPath expressions generated from the formulated conditions to corresponding SQL queries over the underlying database. It is invoked by the `selectMaterialization` function.

Result Retriever Module: This module is invoked after the user clicks "Run". It retrieves the relevant subtrees from the database whose identifiers matches with those in table T. This can be done by joining table T with the table(s) in the database containing the shredded XML document(s). In our system, T is joined with the `PathValue` table [8] containing root-to-leaf paths, their corresponding data values, and order encodings. Reconsidering the above example, this module will return the first `book` subtree of Figure 3 as query results.

The Result Constructor Module: Upon successful execution of the above retrieval, the result can be either displayed as an XML tree or document.

IV. Performance Summary

XBLEND prototype is developed in Java JDK 1.6 and the visual interface is implemented using Java Swing platform and Netbeans visual library. The underlying XML database system is XCALIBUR [8] running on top of Microsoft SQL Server 2005 Developer Edition on an Intel Pentium 4 3.0 GHz system with 3GB RAM. We used 1GB XBench DCSD [9] as synthetic dataset and XML representations of SWISSPROT, PDB, INTERPRO, and *HinvDB* (`hinvdb.ddbj.nig.sc.jp`) as real datasets. The features of these datasets are given in Figure 5. We chose seven queries as shown in Figure 6. $Q1 - Q3$ are XPath expressions on the synthetic dataset whereas $Q4 - Q7$ are XQueries on the real-life datasets. We compare the performance of XBLEND with the XML supports of SQL Server 2005 (denoted as XMSSQL) and IBM DB2 $v9.5$ (denoted as XDB2).

Six unpaid volunteers having moderate knowledge about XML participated in this study. They were briefed on the purpose of the experiment. The task was demonstrated and a sequence of warm-up trials was given prior to testing. Each query was formulated six times by each participant. Note that the faster the user formulates a query, the lesser time we have for prefetching ("worst" case scenario). Consequently, the users were trained to formulate the queries as fast as possible.

ID	Query	Result Size
Q1	/catalog/item[description="Value2"]/title	97,803
Q2	/catalog/item[title="Value1" and description="Value2"]/attributes	50
Q3	/catalog/item[authors/author/contact_information/mailing_address/name_of_country = publisher/contact_information/mailing_address/country/name or (attributes/number_of_pages > 100 and date_of_release = '%2006%' and subject = '%SCIENCE%' and authors/author/date_of_birth = '%-01-%')]	6818

Q4	for $entry in doc("swissprot")/uniprot/entry
	where $entry/@created = "%2005%" and $entry/gene/name = "%aroD%"
	return $entry
		1

Q5	for $entry in doc("swissprot")/uniprot/entry,
	    $interpro in doc("interpro")/interprodb/interpro[@id = $entry/dbReference/@id]
	where $entry/@created = "2005" and $interpro/pub_list/publication/journal = "Bioinformatics"
	return $entry/keyword
		34

Q6	for $entry in doc("swissprot")/uniprot/entry, $interpro in doc("interpro")/interprodb/interpro[@id = $entry/dbReference/@id], $cellcategory in doc("pdb")/PDBx:datablock/PDBx:cellCategory[PDBx:cell/@entry_id = $entry/dbReference/@id] where $entry/keyword = "3D-structure" and ($interpro/pub_list/publication/journal = "Cell" or $interpro/pub_list/ publication/journal = "Cancer Cell") return $entry/gene	2
Q7	for $entry in doc("swissprot")/uniprot/entry, $interpro in doc("interpro")/interprodb/interpro[@id = $entry/dbReference/@id] let $cellcategory := doc("pdb")/PDBx:datablock/PDBx:cellCategory[PDBx:cell/@entry_id = $entry/dbReference/@id] let $hinv := doc("HInvDB")/H-inv/LOCUSXML[CLUSTERID = $entry/dbReference/@id] where $entry/gene/name = "aroD" and (count($cellcategory) > 0 or count($hinv) > 0) and $interpro/pub_list/publication/journal = "Bioinformatics" return $entry/organism	0

Fig. 6. Query set.

Query	QFT	T_w	XMSSQL	XDB2
Q1	10,121.60	15,847.00	90,639.34	230,256.12
Q2	14,549.67	41,945.00	160,488.42	60,096.04
Q3	58,458.00	70,237.33	DNF	19,332.68
Q4	17,860.00	21,766.00	DNF	81,272.21
Q5	34,620.70	34,761.00	NS	81,898.35
Q6	70,674.00	32,646.70	NS	307,906.45
Q7	71,361.30	10,485.00	NS	115,847.87

Fig. 7. Performance (in msec).

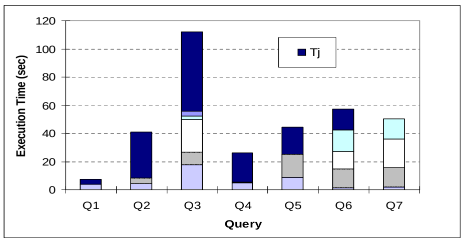


Fig. 8. Cost of partial query fragments.

The reading of the first formulation of each query is ignored. Figure 7 shows the average *query formulation time* (QFT) of each query in XBLEND. The QFT is the duration between the time the first condition is added and the execution of the "Run" command.

The experimental results, shown in Figure 7, confirm the strengths of XBLEND. The symbol "DNF" denotes that the query did not finish execution in one hour and "NS" indicates that the particular query is not supported by XMSSQL. T_w refers to the duration between the time a user presses the "Run" icon to the time when the user gets the query results. Observe that although XDB2 performs better than XMSSQL in most cases, it is slower than XBLEND in most cases. A user can get the query results in XBLEND within $70s$ after he/she presses the "Run".

Figure 8 shows the execution cost of SQL queries to retrieve partial results. Each section of the stacked columns represents the running time associated with the SQL query. For example, for $Q2$ three SQL queries, including the final join query (denoted as T_j in the figure) were issued during prefetching. Therefore, we have three sections in the stacked column. Note that $Q7$ does not execute the join query as it does not have any results that satisfies the query. Observe that the cumulative cost is significantly lesser (except for $Q3$) than the cost of executing the entire query in normal mode (without prefetching).

V. RELATED SYSTEMS

Closest to our system is the effort by Polyzotis et al. [6] in *speculative* query processing in the context of relational data where the final query (or sub-queries that will be present in the final query) is predicted based on the user's usage profile. In comparison, our approach employs deterministic prefetching without speculating on the final form of the query. Speculation can lead to execution time penalties when the prediction is incorrect. In our case, this problem does not arise as we do not predict a user's actions. Furthermore, we do not need to keep track of user's usage profile.

XML caching techniques [4] operate on the final query and do not take into account the individual steps in query formulation. On contrary, our approach materializes partial queries at each formulation step by utilizing the latency offered by GUI-driven query formulation. Hence, in our approach every query benefits whereas in caching *only queries whose results have been cached* improve in performance.

VI. DEMONSTRATION

We demonstrate the power of XBLEND by presenting actual scenarios in which prefetching-based query evaluation is used. Specifically, we will show the followings.

Features and usefulness of XBLEND: We will show specific examples to illustrate user-friendliness in formulating XML queries and show how prefetching-based query processing significantly improves evaluation of complex XML queries compared to normal query evaluation (without prefetching). Users can also state their own ad-hoc queries.

Application of XBLEND **in life sciences**: Querying multiple biological data sources has been a subject of much research in the field of life sciences. We show how XBLEND can be used to easily construct such queries and how it substantially improves query response time.

Robustness of XBLEND: We shall show that XBLEND is beneficial even when errors are committed by users during query formulation [2], [7]. We will demonstrate with specific examples that such errors *do not* have significant adverse impact on the query response time.

REFERENCES

[1] S. ABITEBOUL, R. AGRAWAL, P. BERNSTEIN ET AL. The Lowell Database Research Self-Assessment. *In CACM*, 48(5), 2005.
[2] S. S. BHOWMICK, S. PRAKASH. Every Click You Make, I Will be Fetching It: Efficient XML Query Processing in RDBMS Using GUI-driven Prefetching. In *ICDE*, 2006.
[3] D. BRAGA, A. CAMPI, S. CERI. XQBE (XQuery By Example): A Visual Interface to the Standard XML Query Language. In *ACM TODS*, 30(2):398—443, 2005.
[4] L. CHEN AND E. RUNDENSTEINER. ACE-XQ: A Cache-aware XQuery Answering System. In *WebDB*, 2002.
[5] G. GOU, R. CHIRKOVA. Efficiently Querying Large XML Data Repositories: A Survey. *In IEEE TKDE*, 19(10), 2007.
[6] N. POLYZOTIS, Y. IOANNIDIS. Speculative Query Processing. In *CIDR*, 2003.
[7] S. PRAKASH, S. S. BHOWMICK, K. G. WIDJANARKO ET AL. Efficient XML Query Processing in RDBMS Using GUI-driven Prefetching in A Single-User Environment. In *DASFAA*, 2007.
[8] B.-S SEAH, K. G. WIDJANARKO, ET AL. Efficient Support for Ordered XPath Processing in Tree-Unaware Commercial Relational Databases. *In DASFAA*, 2007.
[9] B. YAO, M. TAMER ÖZSU, N. KHANDELWAL. XBench: Benchmark and Performance Testing of XML DBMSs. *In ICDE*, 2004.