MUSTBLEND: Blending Visual Multi-Source Twig Query Formulation and Query Processing in RDBMS

Ba Quan Truong and Sourav S Bhowmick

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

Abstract. Recently, in [3, 9] a novel XML query processing paradigm was proposed, where instead of processing a visual XML query *after* its construction, it *interleaves* query formulation and processing by exploiting the latency offered by the GUI to filter irrelevant matches and prefetch partial query results. A key benefit of this paradigm is significant improvement of the *user waiting time* (UWT), which refers to the duration between the time a user presses the "Run" icon to the time when the user gets the query results. However, the current state-of-the-art approach that realizes this paradigm suffers from key limitations such as inability to correctly evaluate certain visual query conditions *together* when necessary, large intermediate results space, and inability to handle visual query modifications, limiting its usage in practical environment. In this paper, we present a RDBMS-based *single* as well as *multi-source* XML *twig* query evaluation algorithm, called MUSTBLEND (**MU**lti-**S**ource **T**wig **BLEND**er), that addresses these limitations. A key practical feature of MUSTBLEND is its portability as it *does not* employ any special-purpose storage, indexing, and query cost estimation schemes. Experiments on real-world datasets demonstrate its effectiveness and superiority over existing methods based on the traditional paradigm.

1 Introduction

Formulating XML queries using XPath or XQuery languages often demand considerable cognitive effort from the end users and require "programming" skills that is at least comparable to SQL [1, 7]. The traditional approach to address this challenge of query formulation is to build an intuitive and user-friendly visual framework [4] on top of a state-of-the-art XML database. Figure 1 depicts an example of such a visual interface. Although query formulation now becomes significantly easier, evaluation of XQuery queries (especially over multiple data sources) on existing XML supports provided by commercial RDBMSs is often slow. To get a better understanding of this problem, we experimented with the datasets and queries in Figure 2[1]. Figure 2(c) shows the query evaluation times on XML-extended relational engines of two popular commercial RDBMS. Due to legal restrictions, these systems are anonymously identified as *XSys-A* and *XSys-B* in the sequel. Observe that most queries either take more than 30 minutes to evaluate (denoted by DNF in the paper) or are not supported by (denoted by NS) the underlying RDBMS. Note that the query evaluation time in a visual querying framework

[1] For the time being, the reader may ignore the bold underlined text and the identifiers in braces.

W. Meng et al. (Eds.): DASFAA 2013, Part II, LNCS 7826, pp. 228–243, 2013.

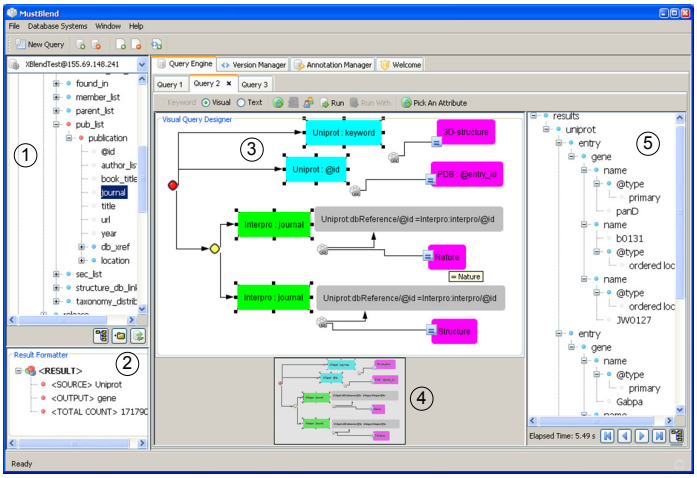


Fig. 1. Visual interface of MUSTBLEND

is identical to the *user waiting time* (UWT), which refers to the duration between the time a user clicks on the "Run" icon to the time when she gets the query results.

A Novel Visual Querying Paradigm. To resolve the issue of unusually long UWT of many XML queries, in [3,9] we took the first step towards exploring a novel XML query processing paradigm on top of a relational framework by *blending* the two traditionally orthogonal steps, namely visual query formulation and query processing. Let us illustrate this paradigm with an example. Consider the XML document in Figure 3(a). Suppose a user wishes to retrieve the `name` elements of entries (`entry/name`) that are related to the *"human"* organism (`organism/name`) and are created (`@created`) in *"2001"*. Using the visual interface in Figure 1, one can formulate the query as follows. (a) *Step 1:* Select the `entry/name` from Panel 1 to Panel 2 as *output expression*. Note that Panel 1 depicts the *structural summary* of the XML data sources. (b) *Step 2:* Select the `created` attribute from Panel 1, drag it to Panel 3, and add the value predicate *"2001"*. (c) *Step 3:* Select the `name` of `organism` from Panel 1, drag it to Panel 3, and add the predicate *"human"*. (d) *Step 4:* Click on the "Run" icon.

If we rely on traditional query processing paradigm, then the query evaluation is only initiated *after* Step 4. Although the final query that a user intends to pose is revealed gradually in a step-by-step manner during query construction (Steps 1 to 3), it is not exploited by the query processor *prior* to clicking of the "Run" icon. In contrast, in the new paradigm query construction and query processing are interleaved to prune false results and prefetch partial query results by exploiting the latency offered by the GUI-based query formulation (processing starts immediately after Step 1).

The key benefits of the new paradigm are as follows. First, since a complex XQuery query is evaluated by a set of smaller queries (to retrieve partial results), this new paradigm is less likely to stress the query optimizer compared to a single complex XQuery in traditional paradigm. Second, it significantly improves the UWT for many

Id	Queries	Result Size	Avg QFT (in sec)
Q1	for $entry in doc('UNIPROT.BIOXML')/uniprot/entry, $interpro in doc('INTERPRO.BIOXML')/interprodb/interpro, $cellCategory in doc('PDB.BIOXML')/PDBx:datablock/PDBx:cellCategory where $interpro/pub_list/publication/year > "1950" {3} and $entry/keyword = "3D-structure"{1} and $interpro/@id = $entry/dbReference/@id {2} and $PDBx:cellCategory/PDBx:cell/@entry_id = $entry/dbReference/@id {4} return $entry/name;	8	46.0
Q2	for $entry in doc('UNIPROT.BIOXML')/uniprot/entry, $interpro in doc('INTERPRO.BIOXML')/interprodb/interpro, $publication in $entry/pub_list/publication where **$publication/journal** = "Structure" {4} and **$publication/year** = "2002" {5} and $entry/@created[contains(., "2001")] {1} and $entry/organism/name = "Human" {2} and $interpro/@id = $entry/dbReference/@id {3} return $entry/name;	23	67.7
Q3	for $entry in doc('UNIPROT.BIOXML')/uniprot/entry, $interpro in doc('INTERPRO.BIOXML')/interprodb/interpro where $interpro/pub_list/publication/journal[contains(., "Cell")] {4} and ($entry/organism/name[contains(., "Mouse")] {1} or $entry/organism/name[contains(., "Human")]) {2} and $interpro/@id = $entry/dbReference/@id {3} return $entry/gene;	8871	44.8
Q4	for $entry in doc('UNIPROT.BIOXML')/uniprot/entry where $entry/organism/name = "Mouse" {1} or $entry/organism/name = "Human" {2} return $entry/protein;	17161	27.4

(a) Representative queries

Source	Size	No. of files	No. of Attributes	No. of Elements
UniProt	1.5GB	1	38,380,645	20,836,316
Interpro	69MB	1	1,427,234	988,079
PDB	287MB	30	692,583	5,578,498

(b) Datasets

Id	XSys-A	XSys-B
Q1	NS	NS
Q2	DNF	NS
Q3	DNF	NS
Q4	68.6	269.9

(c) Query performance (in sec.)

Fig. 2. Query evaluation times of representative queries

queries. Since we initiate query processing during query construction, UWT is the time taken to process a part of the query that is yet to be evaluated (if any).

Related Work and Motivation. Despite these appealing benefits of the new paradigm, the approach presented in [3,9] suffers from the following limitations. Firstly, it was designed only for queries in which every condition a user draws on the query canvas need to be processed *independently*. For example, the conditions drawn in Steps 2 and 3 in the above query can be independently matched against the database and the final query results can be computed by identifying common nodes in the partial results of these two conditions. However, this framework fail to correctly handle queries where conditions may need to be evaluated *together*. Consider the XML document in Figure 3(b). Suppose we wish to retrieve the names of proteins (`interpro/name`) that appear in the *"Nature"* journal (`journal` element) in *"2000"* (`year`). Independent evaluation of these two conditions as above will return the rightmost `interpro/name` element ("*Carboxyl transferase*"). However, it is associated with two *different* `publication` elements instead of a single one containing *"Nature"* and *"2000"*. Hence in order to retrieve correct results, these two conditions must be evaluated *together*. Secondly, [3, 9] retrieves and materializes entire subtrees satisfying matching conditions drawn by users. However, this may adversely affect the overall prefetching performance in many cases due to the size of intermediate results. Thirdly, the new paradigm should be efficient and robust even when modifications (e.g., deletion or update of conditions) are committed by users during query formulation . Systematic investigation of how it handle such query modifications was beyond the goal of the aforementioned study. In this work, we seek to overcome these central limitations by proposing a novel algorithm called MUSTBLEND on top of a relational framework.

2 Visual Twig Query Model

We begin by introducing the twig query model which we support in this paper and the visual interface to formulate such queries.

2.1 Multi-Source Twig (MUST) Pattern

Most XML processors, both native and relational, have overwhelmingly focused on *single-source* AND-twig queries modeled as a twig pattern tree [6]. A *single-source* twig query is evaluated on a set of documents represented by a single XML schema or DTD. Jiang et al. [8] extended the notion of such AND-twig queries to process twigs with both AND and OR operators. Hence, at the very least, our query model should support such queries. Additionally, as discussed in Section 1, our query model should support queries over multiple data sources using joins. We refer to such twig queries as *multi-source twig (*MUST*) patterns*.

A MUST pattern Q is a graph with four types of nodes: location step query node (`QNode`), logical-AND node (`ANode`), logical-OR node (`ONode`), and return node (`RNode`). Each Q has a single node of type `RNode` which represents the output node. While labels of `ANode` and `ONode` are always "AND" and "OR" respectively, `QNodes`' and `RNodes`' labels are tags. An edge in Q can be of two types, namely, *axes edge*

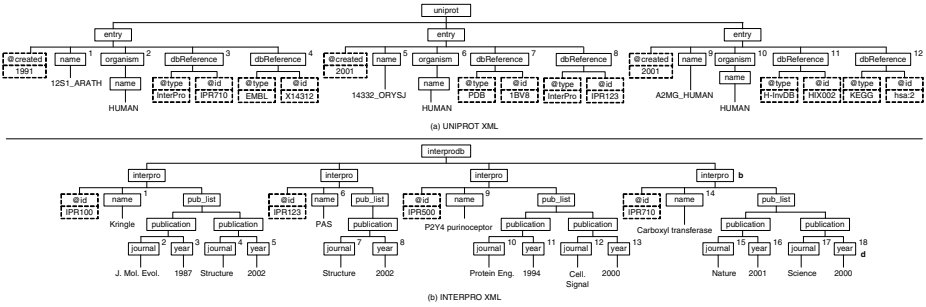
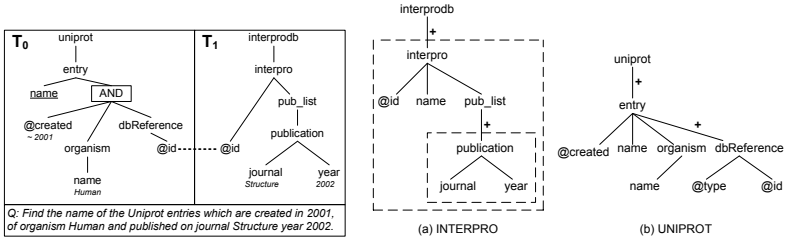


Fig. 3. XML representations of UNIPROT and INTERPRO data sources



(a) A MUST pattern (b) XML structural summaries.

Fig. 4. Twig query model and structural summary

and *join edge*. The former represents parent-child or attribute relationship [2] between a pair of nodes belonging to the same source whereas the latter connects two nodes from two different sources. Specifically, a *join edge* (q_1, q_2) asserts that q_1 and q_2 have equal value[3]. For example, Figure 4(a) shows the MUST pattern representation of the query $Q2$ in Figure 2(a). We denote the `RNode` by underlined tag (*e.g.,* `name`); and axes and join edges as direct and dashed lines, respectively.

Representing MUST Pattern Using XQuery. Observe that the aforementioned MUST pattern can be represented as an XQuery query. A MUST query Q is a 3-tuple $(\mathcal{F}, \mathcal{W}, \mathcal{R})$ where $\mathcal{F}$ is a set of `for` clause items, $\mathcal{W}$ is a set of predicates in DNF in the `where` clause, and $\mathcal{R}$ contains the output expression specified in the `return` clause. Specifically, the syntax of Q is as follows.

$$\begin{array}{rl} \texttt{FOR} & \$x_1\ in\ p_1, \ldots, \$x_n\ in\ p_n \\ \texttt{WHERE} & (a_1 \wedge a_2 \wedge \ldots \wedge a_k) \vee \ldots \vee (c_1 \wedge c_2 \wedge \ldots \wedge c_m) \\ \texttt{RETURN} & r \end{array}$$

We categorize the *where-expressions* in $\mathcal{W}$ into two types, namely *join expressions* and *non-join expressions*. A *join expression* captures the join edge in a MUST pattern and involves predicates expressing join conditions over two document sources. On the other hand, a *non-join expression* expresses a filtering condition on a single document source. In the sequel, we refer to each expression in $\mathcal{W}$ as *condition*. Finally, the `return` clause has a single *output expression* r (`RNode`).

Extension of Query Model. The MUSTBLEND framework can easily support a variety of XPath axis and qualifiers as long as the underlying XML engine can support their evaluation. For instance, if a user visually specifies a path expression containing `AD` and `preceding` axis at a particular formulation step, then this visual action will be translated to a corresponding SQL statement by MUSTBLEND and forwarded to the underlying query engine for execution. Having said this, we would like to stress that a wide variety of XML queries are not easy to formulate even visually as it requires a deep understanding of the language which many end-users do not possess. It is of paramount importance to balance expressiveness and usability in MUSTBLEND as compromising the latter will render it impractical to end-users in a wide variety of domains [7].

2.2 Visual Query Interface

Figure 1 depicts the screen dump of the current version of the *path-based* visual interface of MUSTBLEND. The left panel (Panel 1) displays the XML *structural summary* (discussed later) of different XML data sources. When the users drag a node from Panel 1, the path expression corresponding to this node is automatically built. To formulate a query, the users first specify the output expression r (return node) by dragging that path expression from Panel 1 and dropping it to Panel 2. The *Visual Query Designer*

[2] We consider XPath navigation only along the `child (/)` and `attribute (/@)` axes. Extension to other navigation axis is orthogonal to the proposed technique.

[3] MUSTBLEND only supports equality join condition but inequality join condition can be supported easily.

panel (Panel 3) depicts the area for formulating query conditions. To build a non-join condition, the users drop a path expression in this panel. A *Condition Dialog* will appear for users to fill in all remaining information (op, val). If the dropped expression's data source is different from the output expression's data source, another dialog will appear for users to build the join edge between the two data sources. The user may drop a new condition on an existing condition in Panel 3 in order to indicate her intention to consider these two conditions together. Otherwise, she may drop the new condition on a blank space to indicate that it is independent of existing conditions. Two or more conditions can be combined using AND/OR (default is AND) connectives. The circular nodes in Figure 1 are color coded to represent AND (red) or OR (yellow) connectives. A satellite view (Panel 4) is provided with zooming functionality for more user-friendliness. The user can execute the query by clicking on the "Run" icon. The *Results View* (Panel 5) displays the query results.

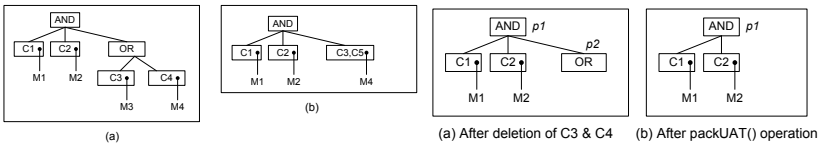
3 Blending Visual Query Formulation and Processing

We now discuss how we can facilitate blending of query formulation and processing. We assume that a user *does not modify* previously constructed query fragments during formulation (no deletion or updates). In the next section, we shall relax this assumption.

Recall that MUSTBLEND GUI provides the flexibility to users to impose constraints on a set of conditions *together* (e.g., conditions on `journal` and `year` elements). However, this feature introduces two challenges. First, it is not always necessary that the underlying query processor need to *evaluate* these conditions together (twig). Hence we need a mechanism to detect automatically when a set of conditions should be evaluated together. Second, in order to facilitate evaluation of these conditions together it is often necessary to identify a common ancestor node (e.g., `publication` element for conditions on `journal` and `year`). It is unrealistic to assume that the end-users should explicitly specify them as it requires understanding of the XML structure. We introduce the notion of *inner structure tree* (IST) and *user actions tree* (UAT) to automatically resolve these two issues. We begin by introducing some auxiliary concepts.

An XML document is modeled as ordered directed trees, denoted by $\mathbb{D} = (\mathcal{N}, \mathcal{S})$, where $\mathcal{N}$ is a set of nodes (elements and attributes) and $\mathcal{S}$ is a set of edges (hierarchical relationships). Given an XML tree $\mathbb{D} = (\mathcal{N}, \mathcal{S})$, a *path* of a node $n \in \mathcal{N}$ in $\mathbb{D}$, denoted as $path(n)$, is a concatenation of dot-separated labels $\ell_1.\ell_2 \ldots \ell_k$, such that $\ell_i (1 \leq i \leq k-1)$ is the label of n's ancestor at level i. ℓ_1 is the label of the root node and ℓ_k is the label of n itself.

We adopt the DataGuide [5] as our XML *structural summary*. Intuitively, a DataGuide *structural summary*, denoted by $\mathbb{S}$, is a tree representing all unique paths in $\mathbb{D}$. That is, each unique path p in $\mathbb{D}$ is represented in $\mathbb{S}$ by a node whose path from the root node to this node is p. An edge may have a label `"+"` *iff* the target node of the edge has cardinality `"+"` with respect to $\mathbb{D}$. Further, every unique label path of $\mathbb{D}$ is described exactly once, regardless of the number of times it appears in $\mathbb{D}$. Figure 4(b) depicts the structural summary of the XML document in Figure 3(b). Observe that the edges incident on `interpro` and `publication` nodes have label `"+"` as `interprodb` and `pub_list` nodes in Figure 3(b) have multiple occurrences of these child nodes, respectively. A subtree of $\mathbb{S}$ is a *Plus-tree* (*P-tree* in short) if its root is the target node of an `"+"`



(a) User actions trees (b) Deletion in a UAT

Fig. 5. User action trees and deletion operation on it

edge. For example, in Figure 4(b) the subtree $g_{publication}$ rooted at `publication` node is a P-tree of $\mathbb{S}$ while the subtree g_{pub_list} rooted at `pub_list` node is a subtree but not a P-tree of $\mathbb{S}$. We denote a set of all P-trees of $\mathbb{S}$ by $ptree(\mathbb{S})$.

Inner Structure Tree (IST). Let $g_1, g_2 \in ptree(\mathbb{S})$ and $root(g_1) \neq root(g_2)$. Then g_1 is an *inner structure tree* (IST) of g_2, denoted by $g_1 \sqsubset g_2$, if and only if g_1 is a subtree of g_2. Note that $path(root(g_2))$ is a prefix of $path(root(g_1))$. For example, in Figure 4(b), $g_{publication} \sqsubset g_{interpro}$ (highlighted by dashed rectangles).

User Actions Tree (UAT). A *user actions tree* (UAT), denoted as U, describes how a set of conditions that are connected by AND or OR connectives are to be processed by MUSTBLEND to generate the final query results. Each internal node of U represents an AND or OR connective. Each leaf node of U is a 2-tuple $v = (\mathcal{C}_a, M)$, where $\mathcal{C}_a$ is a set of non-join conditions that are processed together and M is the temporary relation that stores the prefetched data satisfying $\mathcal{C}_a$. For example, Figure 5(a) depicts two UATs. Observe that M_4 in Figure 5(a)(b) is generated by evaluating C_3 and C_5 together whereas M_2 is generated by processing C_2 independently from rest of the conditions. When do we process a set of conditions together? We elaborate on this now.

Given a condition C, let $target(C.S.exp)$ refers to the *target node* (rightmost node) in the path expression S of C. When the exp is obvious from the context, we denote it as $target(C)$. If $target(C)$ is contained in $g \in ptree(\mathbb{S})$ then we say that g *includes* C, denoted by $C \vdash g$. For example, consider C_3 in $Q2$. Here $target(C_3)$ is the `journal` node that is contained in $g_{publication}$. Hence, $C_3 \vdash g_{publication}$. Given a set of conditions $\mathcal{C}_a$ and $g \in ptree(\mathbb{S})$, g *minimally includes* $\mathcal{C}_a$, denoted by $\mathcal{C}_a \vdash_m g$, *iff* $\forall C_i \in \mathcal{C}_a, C_i \vdash g$ and there $\nexists\, g' \in ptree(\mathbb{S})$ such that $g' \sqsubset g, \forall C_i \in \mathcal{C}_a, C_i \vdash g'$.

Let $\mathcal{C}$ be a set of conditions and r be the output expression on $\mathbb{S}$. Then the conditions in $\mathcal{C}$ are processed together *iff* (a) the label of the parent node of $\mathcal{C}$ in U is `AND` and (b) $g_1 \sqsubset g_2$ where (g_1,g_2) $\in ptree(\mathbb{S})$, $\mathcal{C}_a \vdash_m g_1$ and $(\mathcal{C}_a \cup \{r\}) \vdash_m g_2$. Note that the $root(g_1)$ is the common ancestor satisfying all conditions of $\mathcal{C}$. For example, consider Figure 5(a). Suppose $target(r.exp)$ is the `name` node. If we formulate two conjunctive conditions on nodes `journal` and `year`, then they can be evaluated together to find `publication` nodes satisfying these conditions. This is because $g_{publication}$ is an IST of $g_{interpro}$, $g_{publication}$ minimally includes the conditions on `journal` and `year`, and $g_{interpro}$ minimally includes the conditions on `journal`, `year`, and `name`.

Algorithm 1. The MUSTBLEND algorithm

```
   Input: Actions on the user interface
   Output: Query results M
 1 Initialize M and user actions tree U;
 2 Initialize queue ℚ;
 3 M_o ← fetchOutputExp(r);
 4 𝒜 ← getGUIAction();
 5 while (𝒜 ≠ "Run") do
 6     if (𝒜 == "Add") then
 7         C_add is the new condition;
 8         C_target is the drop target;
 9         _SQL ← fetchCondMatch(r, C, C_add, C_target);
10         U.insert(C_add);
11         ℚ.insert(_SQL);
12     else
13         if (𝒜 == "Delete") then
14             C_del is the deleted condition;
15             U ← deleteHandler(C_del, U, ℚ);
16         else
17             if (𝒜 == "Update") then
18                 C_old and C_new are old and new conditions;
19                 Initialize upFlag = ∅;
20                 U ← updateHandler(C_old, C_new, upFlag, U, ℚ);
21     𝒞.insert(C_i);
22     𝒜 ← getGUIAction();
23 if (ℚ ≠ ∅) then
24     Wait for materializing all partial results;
25 else
26     Modify U by removing unnecessary internal nodes;
27     M ← retrieveFinalResults(U,r);
28 return M
```

3.1 Algorithm MUSTBLEND

We now present the Algorithm MUSTBLEND (Algorithm 1). Importantly, for the sake of generality, we present a generic approach that is independent of any specific relational approaches. The reader may refer to [11] for an example of how various subroutines in the algorithm can be realized on a specific tree-unaware XML storage system. First, when the output expression r is dragged into Panel 2, it materializes the *identifiers* of the elements/attributes in the XML tree that satisfy r by invoking the *fetchOutputExp* procedure (Line 03). It generates an SQL query for this task. An *identifier* of an element n in an XML tree $\mathbb{D}$ is one or more attributes of n that can uniquely identify n in $\mathbb{D}$. Note that we materialize the identifiers instead of entire subtrees because it is more space-efficient. It is worth mentioning that the identifier scheme is not tightly coupled to any specific system as any numbering scheme (*e.g., region encoding*, *dewey number-based* [6]) that can uniquely identify nodes in an XML tree can be used as an identifier.

Next, Lines 05–22 are executed repeatedly until the "Run" icon is clicked. When a user drags a new query condition C_{add} and drops it on an existing condition C_{target}, Lines 07–11 are executed. The algorithm invokes the *fetchCondMatch* procedure to materialize the identifiers in M_o that satisfy C_{add} (Line 9). Then, it adds C_{add} into the UAT U (Line 10). Figure 5(b) depicts the UAT generated after the conditions in the running query are visually formulated. MUSTBLEND detects that C_3 and C_5 need to be processed together and identifies the common ancestor. Lines 13–20 are executed if

the user modifies a portion of the query that have already been constructed. We shall elaborate on these steps in Section 4. Note that the translated SQL queries generated by these steps are inserted into a queue $\mathbb{Q}$. These queries are then processed sequentially in another process thread. The node identifiers retrieved by the above steps are materialized in a set of relations where the schema of each relation contains only document identifier and node identifier attributes.

Once all the conditions are visually formulated, the user may click on the "*Run*" icon to retrieve the query results. Once the materialization of all partial results are completed, the algorithm invokes the *retrieveFinalResults* (Line 27) which traverses U to retrieve the query results from the temporary tables storing the partial results.

***fetchCondMatch* Procedure.** Let $\mathbb{S}_1$ and $\mathbb{S}_2$ be the structural summaries of the data sources of the output expression r and the new condition C_{cur}, respectively. First, this procedure retrieves the P-trees g_1 and g_2 that minimally includes the two input condition sets ($\{r, C_{cur}, C_{target}\} \vdash_m g_1$, $\{C_{cur}, C_{target}\} \vdash_m g_2$). Next, it determines if join across data sources is needed by comparing the data sources of r and C_{cur}. If join is not required ($\mathbb{S}_1 = \mathbb{S}_2$) then it first retrieves the set of conditions $\mathcal{C}_a$ that have already been formulated by the user and $C \vdash g_2 \; \forall C \in \mathcal{C}_a$ and $C_{target} \in \mathcal{C}_a$. If $g_2 \sqsubset g_1$ then it generates an SQL statement that processes the conditions in $\mathcal{C}_a$ and C_{cur} together due to reasons discussed earlier. Otherwise, it first generates SQL statements to retrieve the node identifiers satisfying C_{cur} and then it appends statements for determining subtrees that contain these identifiers as well as satisfy r. Note that when the user drops C_{cur} on a blank space, then $C_{target} = \varnothing$. In this case, g_2 is set to $\mathbb{S}_2$. Consequently, the condition $g_2 \sqsubset g_1$ is not satisfied and the above step is followed to process the new condition independently. When join across data sources is required, this procedure first updates g_1 where $\{C_j, C_{cur}, C_{target}\} \vdash_m g_1$. Then an SQL query is generated for prefetching portion of data satisfying C_{cur} using the join condition C_j. Due to space constraints, the formal description of the algorithm is given in [11].

***retrieveFinalResults* Procedure.** This procedure can be divided into two main steps: (a) processing of the UAT and (b) retrieval of *complete* subtrees from the database satisfying the query [11]. The objective of the first step is to retrieve all identifiers of instances of r that satisfy the set of conditions in the UAT. After that, the second step is used to build an SQL query to extract all subtrees satisfying the identifiers extracted from the first step. While the second step is straightforward, we propose a *disk-based* and *main memory-based* strategies called DISKRETRIEVE and MEMRETRIEVE, respectively, to realize the first step. The DISKRETRIEVE strategy processes U recursively and *returns an* SQL *query* to retrieve the identifiers from the materialized relations. Given the node $root$ in U, the algorithm first identifies whether it is an "AND" node or an "OR" node. If it is an "AND" node, then it adds the "INTERSECT" operator into the SQL statement. Otherwise, the "UNION" operator is used. Then, it retrieves the child nodes of $root$ and processes them one by one. If the child node is a leaf node, then the algorithm adds corresponding SQL statement. Otherwise, it recursively process the internal nodes and finally returns an SQL query for execution. For example, reconsider the UAT in Figure 5(b). The SQL query generated by this procedure is as follows: `select * from M1 INTERSECT select * from M2 INTERSECT select * from M4.`

While the DISKRETRIEVE strategy requires the partial results to be materialized in the database and retrieved the final results using SQL queries, the MEMRETRIEVE approach reduces I/O cost by storing the identifiers in memory. In particular, it stores the partial results in the main memory[4] and use a similar procedure to the aforementioned algorithm except that it retrieves the intermediate relations directly from the memory instead of building SQL queries.

Remark. Observe that Algorithm MUSTBLEND does not exploit predicate selectivities to optimize prefetching performance. Unfortunately, this strategy is ineffective here as users can formulate low and high selective conditions in any arbitrary sequence of actions. Consequently, it is not advantageous to speculate an end-user's subsequent actions in order to take full advantage of selectivity estimates.

4 Visual Query Modifications

In this section, we address the issue of modification to a visual query. We consider two types of modification, namely *delete* and *update*. Deletion enables a user to delete a query condition $C_{del} \in \mathcal{C}$ that has been constructed by him. The update operation allows a user to update a previously formulated condition or change the default AND connective to OR. Specifically, we allow the following updates types (a) Update of the value of a condition. (b) Update of the operator of a condition. (c) Update of AND/OR connectives. Note that the path expression of a condition is not allowed to be updated visually as it often demands syntactic knowledge of XPath expressions from the users. To modify the path expression, one must delete the condition and add a new one.

Handling Deletions. The *deleteHandler* procedure handles deletion of a condition C_{del} in the following way. First, it checks if the translated SQL query for C_{del} is still in the query queue $\mathbb{Q}$. If it is, then it indicates that the query has not been executed yet. Hence, the algorithm will remove it from $\mathbb{Q}$. Otherwise, the results of C_{del} have already been materialized in a temporary table M_C. Consequently, the algorithm will drop M_C. Next, it updates the UAT U by deleting C_{del} from it. Finally, it checks if an internal node of U has become a leaf node due to the deletion of C_{del} and modify U accordingly (*packUAT* procedure). For example, consider the UAT in Figure 5(a). Figure 5(b)(b) depicts the structure of the UAT after deleting C_3 and C_4. Note that if $C_{del} \in \mathcal{C}_a$ (where $1 \leq i \leq |\mathcal{C}_a|$) is deleted, then all conditions in $(\mathcal{C}_a - C_{del})$ shall be reevaluated. The algorithms in Section 3.1 can be exploited for this purpose.

Handling Updates. We first discuss updates on conditions (leaf nodes in the UAT) and then present the effect of updates on the internal nodes. Suppose that a user updates the condition $C_{old} \in \mathcal{C}$ to C_{new}. Let M_{old} and M_{new} be the materialized tables satisfying C_{old} and C_{new}, respectively. There are four possible cases as follow for such update operation. **(a)** ***Case 1:*** $M_{old} \subset M_{new}$. In this case the results in M_{old} also satisfy C_{new}. However, not all nodes satisfying C_{new} have been retrieved. Hence, it is necessary to retrieve these additional nodes and merge them with M_{old}. **(b)** ***Case 2:*** $M_{old} \supset M_{new}$. Nodes satisfying C_{new} are already in M_{old}; however M_{old} also contains nodes that do

[4] The intermediate relations are implemented using *HashMap*.

not match C_{new}. Consequently, these nodes need to be deleted. **(c)** ***Case 3:*** $(M_{old} \cap M_{new}) \neq \emptyset$ *and* $(M_{old} \neq M_{new})$. This case represents the scenario where some of the nodes in M_{old} are part of the result matches for C_{new}. Note that M_{old} also contains nodes that are not relevant to C_{new}. Hence, we need to delete non-matching nodes from M_{old} and retrieving matching nodes that are not in M_{old}. **(d)** ***Case 4:*** $(M_{old} \cap M_{new}) = \emptyset$. We delete M_{old} and retrieve matching nodes for M_{new}.

The *updateHandler* procedure first determines whether the update operation is on a query condition or on an AND/OR node. If the former is true then it determines the *update code* (1, 2, and 0 for Cases 1, 2, and 3 and 4, respectively) based on C_{old} and C_{new} only. In case, it is not possible to determine the code (e.g., the value of a condition is a string) then by default it is considered as Case 4. If the *update code* is 0, then it considers this modification as deletion of C_{old} and insertion of C_{new}. Note that if $C_{old} \in \mathcal{C}_a$ then we execute these two steps as well. If the *update code* is greater than 0, then the algorithm first checks if the SQL query for C_{old} is still in $\mathbb{Q}$. If it is still in $\mathbb{Q}$, then C_{new} will be translated into an SQL query by using the algorithms discussed in the preceding section and it will replace the old query in $\mathbb{Q}$ with the new one. On the other hand, if the SQL query for C_{old} has already been executed, then the algorithm will generate an `INSERT` SQL statement (for Case 1) or a `DELETE` statement (for Case 2). Note that the former statement retrieves additional nodes from the database that satisfy C_{new} and inserts them in M_{old}. Similarly, the latter statement deletes nodes in M_{old} that do not satisfy C_{new}.

Now consider the update of an AND node (recall that it is created by default) to an OR node. If each child leaf node n of an updated AND node represents a single condition C then there is no modification to the prefetching process during query formulation. However, if at least one of the child node n contains two or more conditions ($\mathcal{C}_a$) that need to be processed together then n needs to be modified along with its prefetched relation (if any). Consequently, the algorithm first removes unnecessary internal nodes (if any) from U that may have resulted due to the update operation. If n is updated to OR node, then it is decomposed into a set of leaf nodes where each node represents a single query condition $C_i \in \mathcal{C}_a$. The prefetched partial results (if any) associated with n is deleted and SQL queries for each C_i where $1 < i \leq |\mathcal{C}_a|$ are generated to prefetch partial results matching each of these conditions. Otherwise, if an OR node is restored back to an AND node then the original leaf nodes are restored. Due to space constraints, the formal description of *updateHandler* is reported in [11].

5 Performance Study

MUSTBLEND is implemented in Java on top of a recently proposed *path materialization-based (*PM*)* [6] XPath processor[5] on relational backend called ANDES [10]. We create two variants of MUSTBLEND (see *retrieveFinalResults* procedure), namely one having DISKRETRIEVE strategy (denoted by MB-H) and another MEMRETRIEVE strategy (denoted by MB-M). All experiments were conducted on an Intel Core 2 Quad 2.66GHz processor and 3GB RAM. The operating system was Windows XP. The RDBMS used was MS SQL Server 2008 Developer Edition.

[5] PM approach has advantages over node-based approach when XML data are schemaless [6].

Query	MB-M	MB-H	XSys-A	XSys-B	Zorba
Q1	0.12	0.24	NS	NS	NS
Q2	0.13	0.25	DNF	NS	1495.8
Q3	26.1	0.73	DNF	NS	171.9
Q4	134.4	0.83	68.6	269.9	0.45
Q5	0.16	0.20	3.2	16.0	0.45
Q6	0.25	0.24	2.0	18.0	0.64
Q7	1.34	0.61	72.6	449.2	0.47
Q8	0.13	0.45	165.5	NS	145.8
Q9	2.16	0.51	DNF	NS	1400.6
Q10	0.16	0.86	NS	NS	NS

(a) User Waiting Times (in sec.)

Query	MB-M	ANDES
Q1	2.14	DNF
Q2	9.39	18.3
Q3	2.18	DNF
Q4	1.26	12.7
Q5	1.83	2.1
Q6	1.29	68.1
Q7	1.66	22.0
Q8	1.38	18.5
Q9	2.91	19.6
Q10	3.37	7.1

(b) TPT vs complete query execution times (in sec.)

Fig. 6. Performance results (DNF – Did Not Finish in 30min; NS – Not Supported)

Query	Approach	Step Out	Step 1	Step 2	Step 3	Step 4	Step 5	TPT
Q1	MB-M	0.10 (179430)	0.28 (6123)	0.32 (156172)	0.24 (9)	**0.09**	-	2.03
	MB-H	-	-	-	-	**0.20**	-	2.14
Q2	MB-M	0.09 (179430)	0.32 (5850)	0.34 (9595)	7.28 (30294)	1.14 (4317)	**0.10**	9.27
	MB-H	-	-	-	-	-	**0.22**	9.39
Q4	MB-M	0.12 (179430)	0.23 (7566)	0.13 (9595)	**134.41**	-	-	134.89
	MB-H	-	-	-	**0.78**	-	-	1.26
Q6	MB-M	0.03 (18093)	0.18 (35)	0.17 (12479)	0.28 (18064)	0.43 (682)	**0.21**	1.30
	MB-H	-	-	-	-	-	**0.20**	1.29
Q9	MB-M	0.08 (171790)	0.15 (5601)	1.14 (130318)	0.96 (70327)	**2.23**	-	4.56
	MB-H	-	-	-	-	**0.58**	-	2.91

Fig. 7. Running times of materialization of partial results (in sec.) for representative queries

We compare our ANDES-based MUSTBLEND implementation with two popular commercial XML-extended relational engines, *XSys-A* and *XSys-B* (see Section 1), realizing traditional query processing paradigm. Appropriate indexes were created for all approaches and prior to our experiments, we ensure that statistics had been collected. The bufferpool of the RDBMS was cleared before each run. We also compare *Zorba* (`try.zorba-xquery.org`), an open-source XQuery processor written in C++ which adopted latest optimization techniques [2]. We do not compare it with [9] as the latter does not correctly support queries that require a set of conditions to be evaluated together (e.g., $Q2$).

Experimental Setup. We use the XML representations of UNIPROT, PDB, and INTERPRO downloaded from their official websites. The features of these datasets are given in Figure 2(b). Since *Zorba* fails to handle large datasets (UNIPROT), we reduce the UNIPROT dataset by a factor of 50 (28MB) so that we can study its performance.

We chose ten single and multi-source twig queries that join up to three data sources. Q_1 to Q_4 are shown in Figure 2(a) and the remaining queries are given in [11] (due to space constraints). These queries are selected based of several features such as result size, number of conditions in the `where` clause, number of data sources, existence of ISTs with minimally inclusive conditions (highlighted in underlined bold), and existence of AND/OR connectives. The subscripts of the labels in curly braces in the `where` clause represent the default sequence of steps for formulation of conditions in MUSTBLEND. Note that if a join and a non-join condition have same subscript then it means that the join condition is formulated immediately after its non-join counterpart and are evaluated together in MUSTBLEND. For example, the sequence of steps of Q_1 is

Query	Sequence	Time	Time	Time	Time	MB-M	MB-H
Q5	[C2, C1]	0.09	0.10	-	-	0.14	0.26
Q3	[C3, J3, C2, C1]	0.89	0.24	0.34	-	25.6	0.77
	[C2, C1, C3, J3]	0.18	0.34	0.82	-	26.5	0.70
	[C3, J3, C1, C2]	0.81	0.15	0.34	-	26.5	0.78
Q10	[C2, C1, J3, J4]	0.17	0.67	1.21	0.18	0.15	0.92
	[C1, C2, J3, J4]	0.79	0.32	0.17	1.21	0.15	0.90
	[C1, J3, J4, C2]	0.80	1.21	0.17	0.32	0.16	0.93

Fig. 8. Effect of query formulation sequence (in sec.)

depicted in Figure 2(a). Note that the join condition J_2 and the non-join condition C_2 share same subscript. That is, J_2 is specified immediately after the formulation of C_2 and are processed together in one step. Unless mentioned otherwise, we shall be using the default sequence for formulating a query.

In order to formulate visual queries, fifteen unpaid volunteers with no prior knowledge of XQuery query language participated in the experiments. Details related to participants' profile is given [11]. Each query was formulated six times by each participant (using the default sequence unless specified otherwise) and reading of the first formulation of each query was ignored. The average query formulation time (QFT) for a query by all participants is shown in the right-most column in Figure 2(a).

Experimental Results. We now present performance results of MB-H and MB-M.

User Waiting Times (UWT). Figure 6(a) shows the average *user waiting time* (UWT) of all approaches. It is computed by taking the average of the UWTs of all participants. In *XSys-A*, *XSys-B*, and *Zorba*, UWT refers to the query execution times. Clearly, disk-based and memory-based variants of MUSTBLEND are significantly faster than approaches based on traditional paradigm in most queries. In particular, MB-M and MB-H are at least two orders of magnitude faster than *XSys-A* or *XSys-B* for queries that join multiple data sources ($Q1 - Q3$, $Q8 - Q10$). Also, MB-H typically has superior performance compared to MB-M especially for queries with larger result size (e.g., $Q3$, $Q4$). Note that UWT of MB-H is less than a second for all queries. Lastly, although we use a much smaller UNIPROT dataset for *Zorba*, surprisingly, MB-M and MB-H are still significantly faster than *Zorba*.

Materialization of partial results. We now report the execution times for materialization of partial results of a set of conditions in a visual query. Figure 7 reports the performance of five representative queries (results for all benchmark queries are available in [11]). Each column labeled $Step\ i$ represents the running time associated with the materialization of corresponding query condition(s) of i-th step (subscript in the label inside curly braces) in the sequence. The last step in MB shown in bold refers to retrieval of entire subtrees satisfying the complete query. The values in parenthesis represent the size of the materialized relations. $Step\ out$ refers to the output expression selection step. In response to this action, MUSTBLEND retrieve all nodes (identifiers) in the database satisfying the output expression. Notably, the only difference between MB-M and MB-H is the last step where final results are retrieved (*retrieveFinalResults* procedure). The preceding steps are identical in both approaches.

We can make the following observations. First, the large size of intermediate results does not adversely affect the UWT. Additionally, retrieving all the node identifiers (can

Id	Original query	Modified query
MQ1	for $entry in doc('UNIPROT.XML')/uniprot/entry where ($entry/keyword = "3D-structure" {C_1} or $entry/keyword = "Calcium" {C_2}) and (**$entry/organism/name[contains (., "Cell")]** {C_3} or **$entry/organism/name = "Mouse"** {C_4}) return $entry/gene	for $entry in doc('UNIPROT.XML')/uniprot/entry where ($entry/keyword = "3D-structure" {C_1} or $entry/keyword = "Calcium") {C_2} and (**$entry/organism/name[contains (.,"virus")]** {C_8} or **$entry/organism/name = "Human")** {C_6} and **$entry/feature/@description[contains(.,"protein")]**{C_7} return $entry/gene
MQ2	for $entry in doc('UNIPROT.XML')/uniprot/entry, $interpro in doc('INTERPRO.XML')/interprodb/interpro where $entry/organism/name[contains(., "Human")] {C_1} and $interpro/pub_list/publication/journal = "Structure"{C_2} and **$interpro/pub_list/publication/year = "2002"** {C_3} and $interpro/@id = $entry/dbReference/@id {$J_2$} return $entry/name	for $entry in doc('UNIPROT.XML')/uniprot/entry, $interpro in doc('INTERPRO.XML')/interprodb/interpro, $publication in $interpro/pub_list/publication where ($entry/organism/name[contains(., "Human")] {C_1} and **$entry/protein/name[contains(., "protein")]**) {C_4} and ($interpro/pub_list/publication/journal = "Structure" {C_2} or **$interpro/pub_list/publication/journal = "Cell"** {C_5}) and **$publication/year > "1980"** {C_6} and **$publication/year <= "2000"** {C_7} and $interpro/@id = $entry/dbReference/@id {$J_2$} return $entry/name
MQ3	for $entry in doc('UNIPROT.XML')/uniprot/entry, $interpro in doc('INTERPRO.XML')/interprodb/interpro, $cell in doc('PDB.XML')/datablock/cellCategory/cell where $entry/keyword = "3D-structure" {C_1} and **$entry/organism/name[contains(., "Human")]** {C_2} and $interpro/@id = $entry/dbReference/@id {$J_3$} and $cell/@entry_id = $entry/dbReference/@id {$J_4$} return $entry/name	for $entry in doc('UNIPROT.XML')/uniprot/entry, $interpro in doc('INTERPRO.XML')/interprodb/interpro, $publication in $interpro/pub_list/publication, $cell in doc('PDB.XML')/datablock/cellCategory/cell where $entry/keyword = "3D-structure" {C_1} and **$entry/organism/name[contains(., "Mouse")]** {C_8} and **$publication/year > "1956"** {C_6} and **$publication/year <= "2000"** {C_7} and $interpro/@id = $entry/dbReference/@id {$J_3$} and $cell/@entry_id = $entry/dbReference/@id {$J_4$} return $entry/name

Fig. 9. Effect of query modifications

be large) satisfying an output expression is feasible as it does not affect the prefetching operations and UWT adversely. Second, MB-M is faster than MB-H when the final result set is small (e.g., $Q1$, $Q2$, $Q10$) whereas MB-H is faster when the final result set is large (e.g., $Q3$, $Q4$). Finally, for the majority of the queries in MB, interestingly, the *total prefetching times* (the total time taken for all prefetching operations, denoted by TPT) are significantly less than the query execution times in *XSys-A*, *XSys-B*, and *Zorba* (Figure 6(a)). This is due to benefits of the new paradigm mentioned in Section 1.

TPT vs complete query execution times. The aforementioned experiments do not demonstrate whether the performance benefit of MUSTBLEND is due to the visual querying paradigm instead of the efficiency of underlying storage scheme of ANDES. In this experiment, we shall shed light on this issue. Specifically, we measure the TPT of $Q1 - Q10$ using MB-H and the execution time of each query in its entirety on ANDES. Note that we did not undertake similar experiments on *XSys-A*, *XSys-B*, and *Zorba* as these systems do not allow us to retrieve and materialize node identifiers as partial result matches. Recall that in MUSTBLEND we only materialize node identifiers in order to minimize intermediate results size. Figure 6(b) reports the performance results. Clearly, in most cases the TPT is significantly lower than the cost of executing an entire query on ANDES. Observe that the UWT (Figure 6(a)) is also significantly smaller than the evaluation time of an entire query on ANDES.

Effect of query formulation sequence. A visual query can be formulated by following different sequence of steps. We now assess the effect of these different sequences on the UWT in MB. Figure 8 lists different formulation sequences for three representative queries (results for all benchmark queries are available in [11]), average times (all participants) to retrieve partial results, and the average UWT. Note that $Q5$, $Q3$ and $Q10$ are on one, two and three data sources. Notably, there are hardly any significant changes in both the prefetch times and the UWT. This is primarily due to the following reasons. Firstly, GUI latency can always be exploited by MUSTBLEND at each step irrespective of the ordering

Query	Type	Sequence of Modifications	Prefetch Time	Avg. UWT	Result Size
MQ1	IM	1. Update C4 to C6 (Case 4)	0.23	1.2	1928
		2. Update C3 to C5 (Case 4)	0.13	1.46	2135
		3. Insert *$entry/protein/name = "Protein"*	0.17	0.05	0
		4. Delete *$entry/protein/name = "Protein"*	-	0.31	2135
		5. Insert C7	0.38	0.41	866
	BM	Same sequence as IM	-	0.42	866
MQ2	IM	1. Update the value of C1 to *"Mouse"* (Case 4)	0.46	0.42	972
		2. Update C3 to C6 (Case 1)	7.36	6.57	1653
		3. Insert C7	1.31	2.76	1638
		4. Update the value of C1 to *"Human"*	0.41	0.69	1401
		5. Insert C5	5.96	3.66	0
		6. Update to *OR* node for C2 and C5	7.92	8.35	2748
		7. Insert C4	0.60	0.97	1740
	BM	Same sequence as IM	-	0.98	1740
MQ3	IM	1. Insert C6	1.33	0.20	2
		2. Insert C7	1.38	0.55	2
		3. Update C2 to C5 (Case 4)	0.40	0.10	1
	BM	Same sequence as IM	-	0.12	1

Fig. 10. Effect of query modification in MUSTBLEND (in sec.)

of the visual steps. Secondly, in any query formulation sequence, each visual step results in evaluation of a simple XPath fragment, which is much faster to evaluate compared to a large chunk of complex XQuery as the former stresses the underlying query processor less.

Effect of query modifications. Figure 9 depicts three representative queries on one, two, and three data sources before and after modifications (denoted by $MQ1$, $MQ2$, and $MQ3$, respectively). In this figure, we highlight the changes in underlined bold. For ease of reference, all unique conditions in the original and modified versions of a query are given unique identifiers (e.g., C_1). In order to simulate real-world scenario, we consider two types of modification scenario, namely *incremental* and *bulk* modifications. In *incremental modification* (denoted by IM), after each modification action we execute the query by clicking on the "Run" icon. Hence, if there are n modifications performed by a user then the query is evaluated n times. On the other hand, in *bulk modification* (denoted by BM), all modifications to a query is first formulated before it is executed. Hence a modified query is executed *only once*.

Figure 10 reports the performances of IM and BM in MB-M. Since deletion of a condition does not require retrieval of new matches, we mainly focus on updates. To simulate real-world scenario, we mix update operations with insertion of new conditions. The sequence of operations performed by a user for a query is recorded in the second column. As a user may insert/update a condition and restore it back later (this modification will not appear in the final modified query) after realizing his mistake, we represent this scenario by inserting (resp. updating) and deleting (resp. update back) query conditions that do not appear in Figure 9 (e.g., the third and fourth modification actions for $MQ1$, first and fourth updates for $MQ2$). We can make the following observations from the results in Figure 10. First, all prefetching activities in IM due to the modifications are completed within few seconds. Second, the UWTs for both IM and BM are significantly faster than traditional approaches. The modified $MQ2$ and $MQ3$ do not return any results in 30 minutes or they are not supported by *XSys-A* and *XSys-B*. The UWTs of $MQ1$ for *XSys-A* and *XSys-B* are $173.7s$ and $866.6s$, respectively. *Zorba* takes $22.1s$ and $2957.7s$ for modified $MQ1$ and $MQ2$, respectively. However, it does not

support $MQ3$. These results clearly demonstrate that MUSTBLEND's performance is not adversely affected by query modifications, highlighting again its strength.

6 Conclusions

Our research sought to understand and provide insights to a new XML query processing paradigm where the latency offered by visual query formulation is utilized to prefetch partial results. We have presented MUSTBLEND - an algorithm to realize this paradigm over relational framework by addressing some of the central limitations of [3,9]. Specifically, it can handle richer variety of queries and only stores *synopsis* of intermediate results to make the overall process space-efficient. As MUSTBLEND does not employ special-purpose storage, indexing, and cost estimation schemes to improve UWT, it can easily be built on top of any off-the-shelf RDBMS. Further, the proposed algorithm ensures that the prefetching activities are completely transparent to the users and their interaction behaviors are not affected by this paradigm. MUSTBLEND has excellent performance for a wide variety of queries. It can also gracefully accommodate modifications to a query during construction. All these features are important for deployment of MUSTBLEND in real-world environment.

References

1. Abiteboul, S., Agrawal, R., Bernstein, P., et al.: The Lowell Database Research Self-Assessment. In: CACM, vol. 48(5) (2005)
2. Bamford, R., Borkar, V.R., et al.: XQuery Reloaded. In: VLDB (2009)
3. Bhowmick, S.S., Prakash, S.: Every Click You Make, I Will be Fetching It: Efficient XML Query Processing in RDBMS Using GUI-driven Prefetching. In: ICDE (2006)
4. Braga, D., et al.: XQBE (XQuery By Example): A Visual Interface to the Standard XML Query Language. In: ACM TODS, vol. 30(2), pp. 398–443 (2005)
5. Goldman, R., Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In: VLDB (1997)
6. Gou, G., Chirkova, R.: Efficiently Querying Large XML Data Repositories: A Survey. In: IEEE TKDE, vol. 19(10) (2007)
7. Jagadish, H.V., Chapman, A., Elkiss, A., et al.: Making Database Systems Usable. In: ACM SIGMOD (2007)
8. Jiang, H., Lu, H., Wang, W.: Efficient Processing of XML Twig Queries with OR−Predicates. In: SIGMOD (2004)
9. Prakash, S., Bhowmick, S.S., Widjanarko, K.G., Dewey Jr., C.F.: Efficient XML Query Processing in RDBMS Using GUI-Driven Prefetching in a Single-User Environment. In: Kotagiri, R., Radha Krishna, P., Mohania, M., Nantajeewarawat, E. (eds.) DASFAA 2007. LNCS, vol. 4443, pp. 819–833. Springer, Heidelberg (2007)
10. Soh, K.H., Bhowmick, S.S.: Efficient Evaluation of NOT-Twig Queries in Tree-Unaware Relational Databases. In: Yu, J.X., Kim, M.H., Unland, R. (eds.) DASFAA 2011, Part I. LNCS, vol. 6587, pp. 511–527. Springer, Heidelberg (2011)
11. Truong, Q.B., Bhowmick, S.S.: MustBlend: Blending Visual xml Query Formulation with Query Processing in RDBMS. Technical Report, `http://www.cais.ntu.edu.sg/~assourav/TechReports/MustBlend-TR.pdf`