Worcester Polytechnic Institute
Digital WPI

Computer Science Faculty Publications | Department of Computer Science

7-2009

Achieving High Output Utility under Limited Resources through Structure-based Spilling in XML Streams

Mingzhu Wei
Worcester Polytechnic Institute, samanwei@cs.wpi.edu

Elke A. Rundensteiner
Worcester Polytechnic Institute, rundenst@cs.wpi.edu

Murali Mani
Worcester Polytechnic Institute, mmani@cs.wpi.edu

Follow this and additional works at: https://digitalcommons.wpi.edu/computerscience-pubs

Part of the Computer Sciences Commons

Suggested Citation
Wei, Mingzhu , Rundensteiner, Elke A. , Mani, Murali (2009). Achieving High Output Utility under Limited Resources through Structure-based Spilling in XML Streams. .
Retrieved from: https://digitalcommons.wpi.edu/computerscience-pubs/24

This Other is brought to you for free and open access by the Department of Computer Science at Digital WPI. It has been accepted for inclusion in Computer Science Faculty Publications by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.

WPI-CS-TR-09-07 July 2009

Achieving High Output Utility under Limited Resources through Structure-based Spilling in XML Streams

by

Mingzhu Wei
Elke A. Rundensteiner
Murali Mani

Computer Science Technical Report Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Achieving High Output Utility under Limited Resources through Structure-based Spilling in XML Streams

Mingzhu Wei, Elke A. Rundensteiner and Murali Mani
Worcester Polytechnic Institute
100 Institute Road, Worcester, MA, USA
{samanwei|rundenst|mmani}@cs.wpi.edu

Abstract

Because of high volumes and unpredictable arrival rates, stream processing systems are not always able to keep up with input data - resulting in buffer overflow and uncontrolled loss of data. To produce eventually complete results, load spilling, which pushes some fractions of data to disks temporarily, is commonly employed in relational stream engine. In this work, we now introduce "structure-based spilling", a spilling technique customized for XML streams by considering the partial spillage of their possibly complex XML elements. Such structure-based spilling brings new challenges. First we devise an algorithm, based on the underlying theory of tree pattern containment relationships, that correctly derives the spilling effects on the query plan efficiently. We also examine how to guarantee to generate an entire result set eventually by producing supplementary results in the clean-up stage. Second we tackle the optimization problem, namely, the selection of the reduced query that maximizes output quality. For this, we develop three alternative optimization strategies, namely, OptR, OptPrune and ToX. The experimental results demonstrate that our proposed solutions consistently achieve higher quality results compared to state-of-the-art techniques.

1 Introduction

Motivation. XML stream systems have particularly attracted researchers' interest recently [5, 8, 11, 14, 17, 24] because of the wide range of potential applications such as publish/subscribe systems, supply chain management, financial analysis and network intrusion detection. While XML stream data enters the system at the granularity of a continuous stream of tokens [9][1], it models possibly complex hierarchical structures. For most stream applications, immediate online results are required, yet network traffic may be unpredictable or spiky. When the arrival rate is very high, stream processing systems may not always be able to keep up with the input data streams - resulting in buffer overflow or uncontrolled data loss. To produce eventually complete results, load spilling, which pushes some fractions of data to disks temporarily, is commonly employed in relational stream engines [16, 23, 27, 28] to tackle this problem. In this work, we now introduce "structure-based spilling", a spilling technique customized for XML streams by considering the partial spillage of complex XML elements. In this context we opt to produce partial results during periods of distress - ideally focusing on the most essential and time-sensitive information. The output of "delta" supplementary returned structures can be postponed to a later time, for instance, when there is a lull in the input stream. To the best of our knowledge, there is no prior work on exploring structure-based spilling in the literature. We now motivate the practicability of structure-based spilling via concrete application scenarios as below.

[1] A token in XML stream can be a start tag, an end tag or a PCDATA item.

Example 1. In online auction environments, sellers may continuously start new auctions during some promotion time. When a customer wants to search for "SLR cameras", all matching cameras and their product information should be returned. Some key portions of the results, such as price and customer ratings, will be sent to be displayed first, which aid customers to make decisions. Many consumers tend to use two-stage process to reach their decisions [13] instead of checking complete product information immediately. Consumers typically identify a subset of the most promising alternatives based on the displayed results. Other product attributes, such as sizes and product features, are often evaluated later after consumers have identified their favorite subset. Thus when the system resources are limited, the query engine may spill unimportant attributes to the disk while producing partial results containing key information such as price and customer ratings.

Example 2. In network intrusion detection systems, XML streaming data may come from different nodes of the wide-area network. We need to analyze the incoming packet information to detect potential attacks. If some packets are dropped, the thrown packets may contain the information related to the attack. In this case, throwing packets directly may lead to a later failure to detect and understand the ins and outs of attacks. Instead pushing unimportant fractions of data into disks temporarily when system resources are limited can avoid such problem.

Example 3. Facebook users may edit their personal profiles and send messages to their friends at any time. Status updates, composed of possibly nested structures including updates from friends, recent posts on the wall and news from the subscribed group, are generated continuously. However, different users might be interested in specific primary updates. For instance, a college student wants to make new friends. He wants to be notified when his friends add new friends. A girl who likes watching pictures hopes to get notified as soon as her friends update their albums. When the system resources are limited, it may be favorable to delay the output of unimportant updates and instead only report "favorite updates" to the end users.

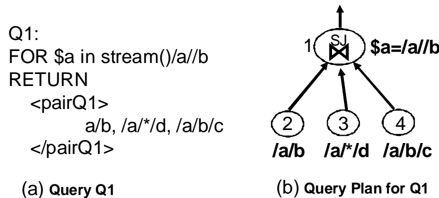


Figure 1. Query Q1 and Its Plan

Challenges. Such structural-based spilling brings new challenges which do not exist in the relational stream. For example, let us look at query Q1 and its plan in Figure 1. Assume we can spill any path in the query when the stream system cannot keep up with the arrival rate. Assume the path $/a/b$ is chosen to be spilled, i.e., all b elements on path $/a/b$ are flushed to disk. Note that multiple paths (e.g. 1, 2 and 4) in the query plan are actually affected (as side effect) by such spilling. Let us refer to the user query after spilling has been applied as *reduced query* and the early output produced by it *reduced output*. To assure the correctness of reduced output, we have to derive the spilling effects on query. This requires us to correctly identify the relationships between the spilled path and other paths in the query. Further, to eventually produce entire yet duplicate-free result set, we need to generate supplementary results correctly. Observe that for each output tuple (e.g. $< pairQ1 >$ in Q1), only partial result structures are produced (e.g. b elements are missing) immediately. Supplementary results can only be generated at a later time when the system has sufficient computing resources with the corresponding reduced output submitted to the output stream long ago. This requires us to design an output model to match supplementary "delta" structures with partial result structures. In addition, to generate supplementary results, we may have to flush extra data to disk to guarantee that the entire result set can be produced. Finally, we have to tackle the optimization problem, namely, the design of the reduced query that maximizes output quality. Our goal is to generate as many high-quality results as possible given limited resources.

Proposed Approaches.

Determine Spilling Effects. To derive the correct spilling effects on the original user query, we need to determine the relationship between the spilled path and other paths in the query. We devise an H^+ algorithm adapted from [22] which is able to determine whether two paths have containment relationship efficiently. To the best of our knowledge, no work has been done to determine such spilling effect due to data dependency over XML stream data.
Complementary Output Model. We propose a complementary output model, which extends from the hole-filler model in [10], to facilitate the matching of the supplementary results with prior generated output. We will illustrate that the hole-filler model naturally fits our spilling scenario.
Produce Correct Supplementary Results. We examine how to flush extra data to generate correct supplementary results. We guarantee to spill a minimum set of data needed for supplementary query execution.
Optimization Strategies. Our final goal is to choose appropriate structures to spill to maximize output quality. We develop three optimization strategies, namely, OptR, OptPrune and ToX. OptR and OptPrune are both guaranteed to identify an optimal reduced query, with OptPrune exhibiting significantly less overhead than OptR.
Contributions. Our contributions are summarized as below:

1. We develop an H^+ algorithm which is able to determine relevant relationships between any two paths efficiently.

2. We propose a complementary output model, which can help supplementary "delta" result structures match reduced output easily.

3. We formulate our structure-based spilling problem into an optimization problem and propose three alternative solutions, OptR, OptPrune and ToX.

4. Our experimental results demonstrate that our approaches consistently achieve higher quality results compared to state-of-the-art techniques.

2 XML Stream Processing Background

Queries We Support. We support the core subset of XQuery in the form of "FOR WHERE RETURN" expressions (referred to as FWR) where the "RETURN" clause can contain further FWR expressions; and the "WHERE" clause contains conjunctive selection predicates, each predicate being an operation between a variable and a constant. A large range of common XQueries can be rewritten into this subset [19].
Query Pattern Tree. The query tree for Q1 is given in Figure 2. Each navigation step in an XPath is mapped to a tree node. We use single line edges to denote the parent-children relationship and double line edges to denote the ancestor-descendant relationship. In this work we assume the spilled path can be any node in the query pattern tree.

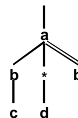


Figure 2. Query Tree for Q1

Algebra Query Processing. We assume the queries have been normalized using the techniques in [6] [2]. Queries are then translated as follows into a plan. For each binding variable in the "for" clause, a structural join is conducted between the binding variable and the paths in the "return" clause. Paths in the "return" clause are translated into

[2] We can use the techniques in [7] to optimize the query.

inputs to the structural join operator. The expressions in the "where" clause are mapped to select operators. Finally a tagging function is on top of the plan taking care of the element construction. Here we focus primarily on the structural join, the core part of the XQuery plan, while tagging is not further discussed. For instance, for the plan in Figure 1, a structural join is conducted between $\$a$ and each of its branches.

Basic Processing Unit (BPU) refers to the smallest input data unit based on which we can produce results independently. It can be a document or a topmost element extracted by the query. When we encounter the end of a BPU in the incoming data, we can produce the result structure. For the query Q1 in Figure 1, the BPU is an a element on path $/a$. When $< /a >$ is encountered, we can produce $< pairQ1 >$ result structures. This provides an efficient way to produce output as early as possible for XML streams [12]. In our work, BPU is the topmost element on the query tree.

3 Basics of Spilling Issues

When to Spill and When to Clean up.

The problem of deciding when the system needs to spill tuples is not a question specific to XML stream while any existing approach from the literature [23, 28] could be employed here. Thus we adopt the following approach for simplicity in our system. We assume a fixed memory buffer to store input XML stream data. As soon as all tokens in an XML element have been processed, we clean those tokens from the buffer. We assume a threshold on the memory buffer that allows us to endure periodic spikes of the input without causing any overflow. During execution, we periodically monitor the memory buffer. When buffer occupancy exceeds the threshold, we trigger the spilling phase. Later on when the arrival speed becomes near zero, we invoke the clean up stage to generate supplementary results.

In this work, we assume single threading, that is, either the reduced query or the supplementary query is running at any time. Parallel execution of the reduced and supplementary queries for slow input rates will be considered in the future.

Complementary Output Model.

In the clean up stage, supplementary results are generated to complement the reduced output. Since partial result structures may be generated for each output tuple, this requires us to design an output model that can efficiently match supplementary "delta" structure with reduced output. Here we propose *complementary output model*, which extends from the hole-filler model [10]. Hole-filler model has been designed to organize out-of-order data fragments when an XML document is split into multiple fragments. Our idea is to explicitly mark a hole in the output element with a unique identifier to indicate missing data. In the later cleanup stage, we produce fillers to fill in these holes, which in our context are supplementary results. The reduced outputs and supplementary results for Q2 when spilling $/a/b$ are shown in Figures 3(c) and (d) respectively (data is shown in Figure 3(b)).

```
Q2: FOR $a in stream("s2")/a
    RETURN <pairQ2> $a//b, $a/d, $a/b/c </pairQ2>
```

To distinguish and match efficiently between holes and fillers, we define three types of IDs, namaly, BPU ID (BID), Result Structure ID (RID) and Path ID (PID). Only fillers and holes with the same IDs can be matched. For instance, the first filler in Figure 3(d) indicates the missing $b1$ and $b2$ for path $/a//b$ in the $< pairQ2 >$ element for the first BPU (a element).

4 Derive Spilling Effects on Query Paths

4.1 Determine Relationships Between Two Paths

As illustrated in Section 1, the spilling of some path may cause other paths to be affected due to data dependency. Hence we need to determine the affected paths in the query. To achieve this, we have to find out the relationships between the spilled path and other paths in the query. Recall that any node in the query tree can be spilled.

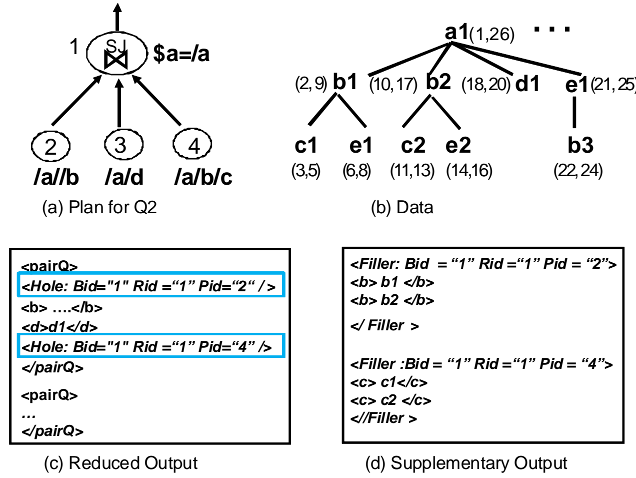


Figure 3. Example for Output Model

Therefore we need to determine the relationships between any two paths so that we can derive the spilling effects correctly.

For a given XPath expression $p \in XP^{\{*,//\}}$ and input document t, we denote by $p(t)$ the set of nodes in t returned by the evaluation of p. Here nodes may correspond to subtrees. For instance, $/a/b$ returns the subtrees rooted at nodes b whose parents are of type a.

We characterize all possible relationships between two paths below:

- **Subset and superset**. Path p is a *subset* of q if q contains p [3] [22]. We also say q is the *superset* of p.
- **Subsumption and subpart**. We call p is a *subpart* of path q if for any node x of path p, we can find a node y for path q where x is a substructure of the subtree rooted at y. Or we say q *subsumes* p. For instance, path $/a/b/c$ is a subpart of $/a/b$ because for any c nodes, their parents must be of type b.
- **Might overlap**. If the nodes of path p might overlap with the nodes of path q in structure, we call these two paths *might overlap*. For instance, $/a/b$ might overlap with path $/a/*/c$ since the node b may have children of type c.
- **Independent**. When no overlapping structures for nodes of two paths exist, we call these two path are *independent*. For instance, path $/a/b/c$ is independent from path $/a/d$ since subtrees rooted at node c cannot overlap with subtrees rooted at d.

The subset (a.k.a. containment) relationship has been studied in [21]. Their algorithm, henceforth called H, decides the containment relationship by looking for homomorphisms between two paths. A homomorphism is a function that matches two paths using the following rules [4]: 1) A label can map to the same label; 2) "*" can map to any label; 3) "//" can map to a subpath that may contain zero or more edges.

The H algorithm iterates over every pair of nodes on path p and q and checks whether there exists a homomorphism from the node on q to a node on p in a bottom up fashion. Table C(x,y) with $x \in$ NODES(p) and $y \in$ NODES(q) is computed for each iteration. C(x,y) is a boolean value denoting whether there exists a homomorphism from the subpath rooted at y to the subpath rooted at x. Figure 4(a) illustrates a homomorphism between

[3] Subset in our context is the same as the containment definition.

[4] In this context, if there is a homomorphism from q to p, then p is contained by q.

$/a/d/e/c$ and $/a/*//c$ in a bottom up fashion. Observe that the bottom node c on path p2 can map to c on p1 based on rule 1. "*" maps to d and "//" maps to edges from d through e until c based on rule 2 and 3 respectively (Table C is shown in Figure 4(b)). Finally c(1,1) returns true. Thus we conclude that $/a/d/e/c$ is a subset of $/a/*//c$.

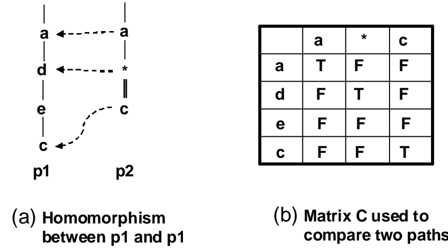


Figure 4. Homomorphism between p1 and p2

In our context, H algorithm guarantees containment checking is sound and complete because the compared paths satisfy $q \in P^{\{*,//\}}$ [22]. However, H algorithm is not sufficient for our goal. Since we aim to compare the relationship of any two paths in the query tree, we hope to avoid duplicate comparisons. For instance, consider two pairs of paths $/a/b/c$ with $/a//b/c$ and $/a/b$ with $/a//b$. If we know $/a/b$ is a subset of $/a//b$, we can reuse this result to judge the relationship between $/a/b/c$ and $/a//b/c$. However, this is not possible for H algorithm since it compares the nodes in a bottom-up manner. In addition, we need to judge other relationships, such as subpart and subsumption. We illustrate now that H algorithm is not able to do so. For instance, consider paths p3 $/a/b/c$ and p4 $/a/b$. The table C is depicted in Figure 5(b). Obviously p3 is a subpart of p4. However, the last b node of p4 cannot find mapping to the node in p3 using H algorithm. Hence c(3,2) is set to false and finally c(1,1) returns false too. We can only conclude that p3 is *not* a subset of p4. But we cannot conclude that p3 is actually a subpart of p4.

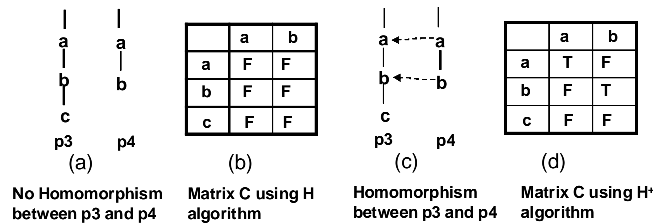


Figure 5. Homomorphism between p3 and p4

To solve the above problem, we instead propose to run H algorithm in a top down manner. Further we change the meaning of C(x,y) as follows: C(x,y) is true if there exists an embedding from the subpattern from root to y to the subpattern from root to x. Since the mapping rules are the same, we can guarantee that we can still find homomorphism if path p1 is contained by p2. Let us call the adapted algorithm the H^+ *algorithm*. We have the following theorem.

Theorem 1 *The containment relationship is derived correctly by the H^+ algorithm.*

The above theorem states that the containment relationship is derived correctly when the H algorithm is run in a top down manner with the new definition of C(x, y) as described above. The proof is omitted due to space limitation.

	/a	/a/b	/a/b/c	/a/*	/a/*/d	/a//b
/a	Eq.	Subsume	Subsume	Subsume	Subsume	Subsume
a/b	Subpart	Eq.	Subsume	Subpart	M.O.	Subset
/a/b/c	Subpart	Subpart	Eq.	Subpart	Ind.	Subpart
/a/*	Subpart	Superset	Subsume	Eq.	Subsume	M.O.
/a/*/d	Subpart	M.O.	Ind.	Subpart	Eq.	M.O.
a//b	Subsume	Superset	Subsume	M.O.	M.O.	Eq.

Table 1. Comparison Matrix for Q1

Table C for comparing paths p3 and p4 using the H^+ algorithm is given in Figure 5(d). Observe that we can conclude that $/a/b/c$ is a subpart of $/a/b$ since there exists a homomorphism from p4 to a subpath of p3, namely, $/a/b$. Another advantage of the H^+ algorithm is that we can reuse the comparison results of any subpaths. Since searching for the homomorphism is conducted in a top-down manner, this means when we compare two paths p and q, we automatically derive the containment relationships between any subpath of path p and any subpath of path q.

To derive the correct subpart relationship, we have the following theorem:

Theorem 2 *Path p is subpart of path q if and only if there exits a homomorphism from q to an absolute subpath of p.*

To determine other relationships such as might overlap and independent, we make use of the techniques proposed in [18].

4.2 Construct Matrix for Comparing Any Two Paths

For each node in the query tree, we can assign a unique ID when we traverse the tree in a DFS manner. To record the relationship of any two paths in the query, we construct a matrix M($|T|$, $|T|$), where $|T|$ indicates the total number of nodes in the query tree. Each value M(x,y) in M indicates the relationship of the path from root to the node of ID x and the path from root to the node of ID y. Since H^+ algorithm can reuse the comparison results, we can simply call H^+ algorithm among the longest paths in the query tree. The comparison matrix for Q1 is shown in Table 1.

4.3 Characterization of Spilling Effects

We have so far examined how to determine relationships between any two paths. Based on these relationships, we categorize the spill effects on paths in a query as follows:

1. *Totally Missing (TAM)*. When the path in the query q is a subset or subpart of the spilled path p, we call q *totally missing* since all its data will be lost. For instance, $/a/b/c$ is TAM due to spilling path $/a/b$.

2. *Partially Missing (PAM)*. When path in the query q is superset of the spilled path, we call *q partially missing* since a subset of its matching data might be lost. For instance, path $/a//c$ is PAM due to spilling $/a/c$, since all the c nodes that have parents of type a will be spilled.

3. *Subpart Missing (SAM)*. When the path q in the query subsumes the spilled path p (i.e. p is a subpart of q), we call q *subpart missing*. For instance, $/a/b$ is SAM when spilling path $/a/b/c$.

4. *Potentially Missing*. If we cannot guarantee that the path will be affected, we call it "potentially missing". For example, if the spilled path is $/a/b$, we say that path $/a/*/c$ is *potentially missing*, since it might overlap with $/a/b$. However, in this work, we take the conservative approach treating it the same as PAM. Note that for spilled path $/a/b$, the spilled data is actually $/a/b//*$ since the spilling always includes its

descendants. We can intersect $/a/b//*$ with $/a/*/c$. Based on that we conclude that c nodes which satisfy path $/a/b/c$ are missing at the worst case. Hence we treat the path which might overlap with the spilled path the same way as PAM.

5 Generate Entire Result Set

5.1 Reduced Query Execution

We now describe how to execute a reduced query based on the knowledge of spilling effects on the paths in the query. We do not block the output as long as it is correct, even though the result structure may be partial. In other words, the reduced query execution should satisfy maximal output property [25]. Therefore we propose the following policies for the execution of the reduced query so that we can produce as much correct output as possible.

- **Affected path in "for" clause**. Let us call the FLWR block where the path q is defined the home FLWR of q. When the binding variable is totally missing (TAM), the structural join between the binding variable and its branches is blocked. This is because the number of bindings is the "loop counter" of invoking the inner FLWR blocks and the structural join between the binding variable and its branch operators. Otherwise, it is unblocked. When the binding variable is partially missing (PAM), the number of bindings may be reduced but we can still produce some output. When the binding variable is subpart missing (SAM), although a subpart of the binding variable is missing, it does not affect the iterations of the "loop counter". Hence SAM on the "for" path does not affect the query result generation.

Example 5.1 *Figure 6 shows three cases when the binding variable is PAM, SAM and TAM respectively. Figure 6(a) shows the case that the binding variable $\$a$ is PAM due to spilling $/a/b$, since path $/a//b$ contains $/a/b$. Figure 6(b) shows the case when $\$a$ is SAM due to spilling path $/a/b/c$. Figure 6(c) shows the case when $\$a$ is TAM.*

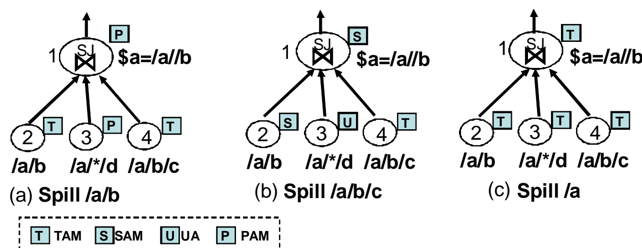


Figure 6. Plan for Q1 with Spilling Effects

- **Non-blocking "return" path**. The structural join is conducted between a binding variable V and all its branches. Based on query semantics, the structural join between a binding variable V and one branch $B(i)$ is independent from the structural join between V and other branches. Therefore a "return" path being affected by spilling does not block the output of other "return" paths in the same FLWR block.

Example 5.2 *For instance, the spilling type of a return path $/a/b$ is SAM due to spilling $/a/b/c$ (Figure 6(b)). Although c children of b nodes are missing, we still allow the output of incomplete structures.*

ID	EqID	For	Return	ID	EqID	For	Return
1	13	SAM	UA	9	11	TAM	UA
2	14	SAM	PAM	10	11	TAM	PAM
3	15	SAM	TAM	11		TAM	TAM
4	16	SAM	SAM	12	11	TAM	SAM
5		PAM	UA	13		UA	UA
6		PAM	PAM	14		UA	PAM
7		PAM	TAM	15		UA	TAM
8		PAM	SAM	16		UA	SAM

Table 2. Possible Combinations Between For Binding and Its Branches

- **Blocking "where" path**. When the "where" path falls into PAM or SAM, this may affect the evaluation of the predicates. In this case, we may not know whether the results should be output or not. Therefore we treat affected "where" paths as blocking. Whenever a "where" path falls into PAM or SAM, the output for its home FLWR and its inner FLWR block cannot produce anything. In the future, we may consider to adaptively handle affected "where" paths. For instance, as long as the path with PAM or SAM evaluates to true, then we allow output. However, we are not focusing on this enhancement in this paper.

5.2 Determine Extra Data to Spill for Supplementary Query Execution

To produce eventually complete results set, we have to generate supplementary results correctly. In this section, we describe how to decide what extra data to spill as well as the supplementary query for generating missing results. Our goal is to spill a minimum set of data needed for supplementary query execution. The eventual result set must be guaranteed to be both complete and duplicate-free.

Since structural join is the core part in the queries we consider, we focus on how to spill extra data to reconstruct the structural join results correctly. Either the "for" path or the "return" path can be of four types, namely, PAM, SAM, TAM or unaffected (UA). Hence 4*4 =16 combinations [5] between the binding variable and branches are listed in Table 2. However, when the "for" binding variable is totally missing (TAM), query processing is blocked. In this case, the entire result set has to be regenerated. Hence we only need to investigate 16-4= 12 cases. In addition, when the binding variable is SAM, query execution is not affected. Hence cases 1,2,3 and 4 can be regarded the same as cases 13, 14, 15 and 16 respectively. We label the equivalent cases in the column "EqID" in Table 2. Clearly, it is not necessary to consider case 13 since complete results are produced in the reduced output. Finally we only need to consider the cases 5-8, 14, 15 and 16. We describe how to derive a minimum set of extra data to flush to disk and how to compute supplementary results below.

5.2.1 Structural Join with Only PAM Type

There are three possible cases for the structural join with PAM (case 5,6 and 14 in Table 2). The first case is that the binding variable falls into PAM but the branch is unaffected (UA). The second case is that the binding variable is UA but the branches are PAM. The third case is that both binding variable and branch are PAM.

Binding variable is PAM and branch is UA.

Now we examine how to generate supplementary results when the binding variable is PAM and its branch operators are all UA. Let us examine an example query Q3.

```
Q3: FOR $a in stream("s3")/a
    RETURN <result>
     /a/b, /a/*, FOR $b in $a//b
           RETURN <p> /a/c, /a/f </p>
          </result>
```

[5] If "where" path is affected, the output is blocked. It is equal to the case that "for" path is TAM.

The plan for Q3 is shown in Figure 7. $\$b$ is PAM when $/a/b$ is spilled. However, op5 and op6 are unaffected since they are independent from path $/a/b$. Here we examine how to generate correct intermediate output of op4. Given the data in Figure 7(b), the output of op4 would generate reduced output (c1, f1) corresponding to b2 since b1 is spilled. The missing result is shown in Figure 7(c). To produce such missing results, we should spill all the data from op5 and op6 beside the spilled data $/a/b$. Therefore we have the following theorem holds.

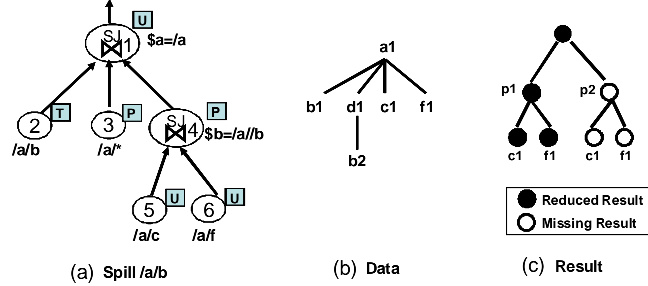


Figure 7. Query Plan, Data and Results for Q3

Theorem 3 *When the binding variable is PAM and all its branch operators are UA, to generate correct missing results for current FLWR block, extra data needs to be spilled is all data from its branch operators.*

The proof of theorem is omitted due to limited space. Now we look at how to generate the supplementary results. We use a superscript m and d to distinguish the data kept in memory and the data flushed to disk respectively. For branch B(i), we have the following equation holds:

$$\begin{aligned} V \bowtie_S B(i) &= (V^d \cup V^m) \bowtie_S B(i) \\ &= (V^d \bowtie_S B(i)) \cup (V^m \bowtie_S B(i)) \end{aligned} \tag{1}$$

The reduced output $V^m \bowtie_S B(1)$, $V^m \bowtie_S B(2)$,... $V^m \bowtie_S B(n)$ have been produced at runtime by conducting structural join between V and $B(1)$, V and $B(2)$,.... V and $B(n)$ respectively. Hence we need to calculate the supplementary results $V^d \bowtie_S B(1)$, $V^d \bowtie_S B(2)$,... $V^d \bowtie_S B(n)$. So the extra data to spill is $B(1)$, $B(2)$, ... $B(n)$.

Since the data might be spilled many times to disk, we use a subscript to indicate the data spilled at different times. Assume that structure V, B(1), ..., B(n) have been pushed k times to disk respectively, meaning the spill data is V_1^d, V_2^d,... V_k^d and $B(1)_1$,... $B(1)_k$,$B(2)_1$,... $B(2)_k$,... $B(n)_1$,... $B(n)_k$ respectively. As we mentioned in Section 2, the results generated based on a basic processing unit are independent from others. In this work we assume always spilling data in batch of basic processing units. So V_x^d does not need to be joined with $B(i)_y$ if x is not equal to y since they do not belong to the same basic processing unit. Therefore, the supplementary join results corresponding to data spilled at time k can be calculated as $(V_k^d \bowtie_S B(1)_k, V_k^d \bowtie_S B(2)_k, ...V_k^d \bowtie_S B(n)_k)$.

Note that case 9 is actually the special scenario of case 5 where $V^m = \emptyset$ and $V^d = V$. So we can use the same equation to calculate the supplementary results.

Binding Variable is UA and Branch is PAM.

In this case, multiple branch operators may fall into PAM at the same time. However, structural join results between V and branch $B(i)$ is independent from the structural join results between V and other branches. The case that one branch operator falls into PAM is considered first and can be easily extended to the case that multiple branches are PAM. For branch $B(i)$, we have the following equation holds:

$$\begin{aligned} V \bowtie_S B(i) &= V \bowtie_S (B^m(i) \cup B^d(i)) \\ &= (V \bowtie_S B^d(i)) \cup (V \bowtie_S B^m(i)) \end{aligned} \tag{2}$$

Obviously, the reduced output $V \bowtie_S B^m(i)$ has been produced at runtime. Now we need to calculate the missing results $V \bowtie_S B^d(i)$. So we have to reconstruct the structural join between V and B and the extra data needed to spill is data corresponding to path V.

Assume V and B has been pushed k times to disk respectively. The missing structural join results between V and $B(i)$ at time k can be calculated as $V_k \bowtie_S B^d(i)_k$.

In the plan for Q2 in Figure 3(a), when path $/a/b$ is spilled, path $/a//b$ is PAM. The structural join between $\$a$ and $/a//b$ can be calculated using Equation 2.

Case 15 is a special scenario of case 14 where $B^m = \emptyset$ and $B^d = B$. Therefore we can use Equation 2 to calculate the supplementary results for case 15.

Binding Variable is PAM and Branch is PAM.

When the binding variable is PAM and the branch is UA, we need to spill all the data from its branch operators. Also based on the case where binding variable is UA and the branch is PAM, we need to spill all the data from binding variable. When the binding variable is PAM and branch is PAM, an intuitive answer is to spill all the data from the binding variable and its branch operators. Can we spill less data? Let us answer this using an example below.

```
Q4: FOR $a in stream("s")/a
     RETURN <result>
      /a/b, FOR $b in $a/*
          RETURN <p> /a/d, /a//b </p>
              </result>
```

The plan and results for Q4 are shown in Figure 8(b). Again we use the data in Figure 7(b). When path $/a/b$ is spilled, b1 is spilled to disk. In this case, the binding variable iteration is reduced from 4 to 3 (b1 is missing). In addition, observe that each result structure is partial. For operator op3, we cannot only spill missing path /a/b without spilling c1, d1 and f1. Because the result structure corresponding to c1, d1 and f1 are still incomplete. On the other hand, for operator op5, we cannot only spill missing path $/a/b$ either since b3 still needs to be joined with the spilled binding value b1. Based on above observations we have the following theorem holds.

Theorem 4 *When the binding variable is PAM and some branch operators are PAM, to generate correct missing results for current FLWR block, we need to spill data for the binding variable and all of its branch operators.*

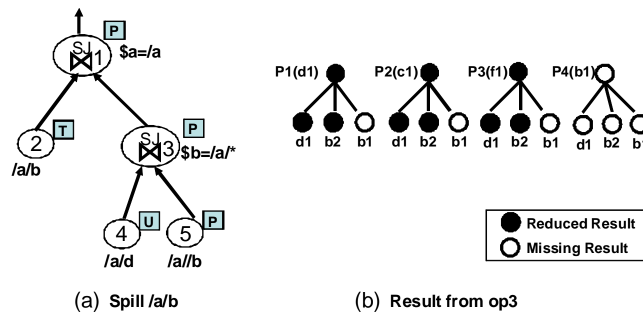


Figure 8. Query Plan and Results for Q4

We omit the proof due to space limitation. Now we investigate how to generate missing results without duplicates. Assume the branch operators falling into PAM are B(s1), B(s2),..., B(se). For each branch operator B(si), we have the following equation holds:

$$
\begin{aligned}
&V \bowtie_S B(si) \\
&= (V^m \cup V^d) \bowtie_S (B^m(si) \cup B^d(i)) \\
&= (V^m \bowtie_S B^m(si) \cup (V^m \bowtie_S B^d(si)) \cup (V^d \bowtie_S B^d(si)) \cup (V^d \bowtie_S B^m(si)) \\
&= (V^m \bowtie_S B^m(si) \cup (V^m \bowtie_S B^d(si)) \cup (V^d \bowtie_S B^d(si)) \cup (V^d \bowtie_S B^m(si)) \\
&= (V^m \bowtie_S B^m(si) \cup V^m \bowtie_S B^d(si)) \cup (V^d \bowtie_S B(si))
\end{aligned} \tag{3}
$$

Clearly, $(V^m \bowtie_S B^m(si)$ has been generated in the reduced output. So we need to calculate other parts in Equation 3.

5.2.2 Structural Joins with SAM Type

Binding variable is UA and branch is SAM.

Multiple branch operators may fall into SAM at the same time. We describe how to generate the structural join results for one SAM branch operator and we can easily extend it to the case that multiple return paths fall into SAM.

When branch operator B(i) falls into SAM but not PAM, we use D to indicate its missing subpart. For all the data corresponding to branch B(i), its subpart D is missing all the time. The missing results can be calculated as: $V \bowtie_S B(i) \bowtie_S D$. Hence we have to reconstruct the structural join between V and B. This requires us to spill all the data of B(i), D and its binding variable V to guarantee the correct missing results is produced. However, since $V \bowtie_S Bi$ includes the output from branch operator Bi, it is already produced in the reduced query execution. We can spill this part of results directly.

When the branch operator Bi belongs to both SAM and PAM, the missing results are divided into two parts. The first part is the missing output corresponding to path $B^d(i)$ (not including subpart D). The second part is the missing output corresponding to subpart D. Based on the conclusion for case 14, the missing results for path $B^d(i)$ are $V \bowtie_S B^d(i)$. For the missing output corresponding to subpart D, we have the the following equation:

$$
\begin{aligned}
&V \bowtie_S Bi \bowtie_S D \\
&= V \bowtie_S (B^m(i) \cup B^d(i)) \bowtie_S D \\
&= (V \bowtie_S B^d(i) \bowtie_S D) \cup (V \bowtie_S B^m(i) \bowtie_S D)
\end{aligned} \tag{4}
$$

Note that $V \bowtie_S B^m(i)$ has been produced at runtime. $V \bowtie_S B^d(i)$ can be calculated in the supplementary query execution.

Binding variable is PAM and Branch is SAM.

Similar to the cases that binding variable is PAM and the branch is SAM, we can combine the equations in these two cases to calculate missing results. The detailed discussion is omitted due to space limitation.

5.3 Completeness and Uniqueness of Final Result Set

Based on the discussions above, we have the following theorem holds:

Theorem 5 *The final result set is complete and no duplicate results will be generated.*

Completeness. First, prove the results satisfy completeness property. Assume a result structure S_i belongs to the output results. But it is not produced in the output. WLOG, we assume S_i is composed of many substructures.

Since the results are generated in one of the following cases: 1) all the substructures of S_i are generated in the reduced query execution; 2) all the substructures of S_i are generated in the supplementary query execution; or 3) some substructures of S_i are generated in the reduced query execution and the other substructures are generated in the supplementary execution.

If condition 1 is satisfied, we can conclude that the tuple S_i must fall into the cases that either both binding variable and the branch are UA or the binding variable is PAM and branch is UA (case 1,5,9 in Table 2). In this

case, the binding variable value for this tuple must belongs to V^m. We have showed that we produce all possible results in the reduced query execution in Section 5.1. If condition 2 is satisfied, we observe that it must fall into the cases where all branches are UA or TAM (cases5,7,9,10,11,12). However, the values of binding variable for tuple S_i must belong to V^d. We have showed in Section 5.2 that we guarantee producing all possible missing results in the reduced query execution. If condition 3 is satisfied, there must exist some branch which falls into either PAM or SAM (cases 6,8,14 and 16 in Table 2). Similarly, we have showed all possible outputs are generated in the reduced query execution and the supplementary query execution.

Based on above discussion, we conclude the assumption that the missing output tuple S_i is not reported by the output is not possible. Thus, our spilling algorithm produces all output results.

Uniqueness. Assume result structure S_i is reported twice in the output. We denoted such two instances S_1 and S_2. Since we only have two phases of query execution, reduced query and supplementary query execution. Thus we identify the following two cases:

Case1: S_1 and S_2 are both generated in the reduced output. As we showed in Section 5.1, this is not possible since the structural join in reduced query would not generate duplicate results.

Case 2: S_1 and S_2 are both generated in the missing output. As we showed in Section 5.2, this is not possible either since the duplicate-free computation is considered in the formula of computing supplementary results.

From the above two cases, we conclude that the assumption that the tuple S_i is reported twice is not valid. □

6 Metrics of Quality and Cost

Our optimization goal is to select optimal structure to spill to maximize output quality at the spilling stage. In this work we focus on maximizing the quality of reduced output. We now describe the metrics of quality and cost for measuring the alternative reduced queries.

6.1 Output Quality

Previous studies on approximate query answering tend to focus on the relational model, where the output quality is usually measured by the output rate or the cardinality [3,26]. However, in our work, since each output result may be partial, measuring the cardinality of the output is obviously not sufficient. Here we propose a "fine-granularity" output quality model which aims to measure the quality of partial XML output results. We measure the quality of the reduced output based on the following factors:

1. **Cardinality**. Since a return structure may be composed of nested substructures, some substructure may only return a subset. So we count the cardinality of each substructure into output quality.
2. **Shape**. Returned substructures may not be of the full shape when the corresponding paths in query fall into SAM. To differentiate such substructures from others, we now define a *shape indicator* to indicate how full each substructure is.

The shape indicator for a path q in query can be calculated as $S_q = \frac{Size\ of\ element\ after\ spilling}{Size\ of\ element\ without\ spilling}$ [6].

When a path falls in SAM, its shape indicator is less than 1. In this sense the quality is "punished " because of returning incomplete substructures.

Recall that the topmost element is the smallest data unit which can produce result structure. We define *unit quality* as the quality gained by executing the reduced query on a topmost element. We measure unit quality using the formula below:

$$\nu = \sum_{n} \sum_{i=0}^{j} \sum_{q \in B(i)} N_q * S_q \quad (5)$$

[6]Here we assume the size of an element is fixed.

Notation	Explanation
N_P	Number of elements matching P for a topmost element
$S_{\bowtie}$	Join Selectivity
C_j	Cost of comparing two elements
$C_{I/O}$	Cost of disk I/O
C_s	Cost of stack operation

Table 3. Notations Used in Cost Model

Here n indicates the number of return structures generated per topmost element. Each returned structure is composed of j substructures. q denotes the type of nodes matching branch $B(i)$. N_q and S_q denote the cardinality and shape indicator of q respectively.

Path	Quality	
	Spill /a/b	Spill /a/b/c
/a//b	1*1	1*1+2*0.5
/a/d	1*1	1*1
/a/b/c	0	0

Figure 9. Quality for Q2

Example 6.1 *We calculate the unit quality of Q2 for data in Figure 3(b) (plan is shown in Figure 3(a)). The quality of each substructure is shown in Figure 9. For each topmost element* a*, a result structure* $< pairQ2 >$ *is returned. When spilling path* $/a/b$*,* $d1$ *and* $b3$ *are returned. The unit quality of the reduced query is 1+1=2. When spilling* $/a/b/c$*,* $/a//b$ *returns three elements,* $b1$*,* $b2$ *and* $b3$*. For* $b1$ *and* $b2$*, their shape indicators are both equal to 0.5 (calculated using Equation 5) since their* c *children are missing. So the output quality for* $/a//b$ *is 1+2*0.0.5= 2. The unit quality is 1+2=3.*

6.2 Cost Model

We now define a cost model for comparing alternative reduced queries. We measure the cost as the average time of processing a topmost element (we call it unit processing cost). We divide the processing cost into the following parts: *Locating Cost* (LC) that measures the cost spent on retrieving data and *Join Cost* (JC) spent on structural joins. In addition, in the spilling stage, since we need to flush data to disk, we call the cost spent on spilling data *Spilling Cost* (SC). Since our goal is to optimize the quality of the reduced query, we focus on the cost model of measuring runtime cost savings for the reduced query. The cost measurement of the supplementary query is not discussed here and is part of our future work.

Locating Cost. Since XML streams can be scanned only once, when the tokens corresponding a path in the query arrive, we must "recognize" them. The locating cost indicates the cost spent on retrieving tokens. Automata are widely used for pattern retrieval over XML streams [8, 17]. The relevant tokens are "recognized" by the automata and then assembled into elements to be further filtered or returned. However we do not need to consider locating cost savings. The reason is that even if we flush some elements to disk, we still have to retrieve them from the input stream.

Join Cost. The main computation savings come from the structural joins. Since we assume stream data arrives in order, the elements for both join inputs are sorted. We can apply an efficient structural join algorithm Stack-Tree-Anc [1] since both inputs are sorted. We use the cost model for this algorithm in [29]. We estimate the cost of structural join using the formula as below :

$$2 * N_V N_{B(i)} S_{\bowtie} C_j + 2 N_V C_s \quad (6)$$

Here N_V and $N_B(i)$ indicate the number of binding variables and branches per topmost element. The notations are in Table 3. Based on Eqaution 6, we can easily calculate structural join savings for the reduced query.

Spill Cost. Although join computation is saved due to spilling, we have to calculate spill costs. Based on our discussion in Section 5.2, we may have to spill other paths to enable future supplementary result generation. Let us use SP to denote the set of paths to be spilled to disk. The flush cost can be calculated as follows:

$$\sum_{q \in SP} N_q C_{I/O} \quad (7)$$

7 Choosing the Optimal Structure to Spill

7.1 Search Space of Spill Candidates

Before solving the problem, we need to first generate all spill candidates. We assume any location and any number of nodes in the query tree can be spilled. Assume there are $|T|$ nodes on the query tree. Then there are $C^0_{|T|} + C^1_{|T|} + ... + C^{|T|}_{|T|} = 2^{|T|}$ possible candidates. An example query tree and its possible candidates are shown in Figure 10. Each node in the lattice represents one candidate. The top node means spilling nothing (i.e. initial query). The bottom candidate indicates spilling everything (i.e. empty query). Each level i lists all candidates spilling i nodes from query tree. Search space scales quickly since it is exponential in the number of nodes in the query tree.

We now observe that we can reduce the search space because some candidates actually result in the same spilling effects. Recall that when we spill data corresponding to path p from the query tree, all its descendants are also flushed to disk. This leads to the following observation:

Observation 7.1 *If a spill candidate includes two nodes which satisfy the ancestor-descendant (or parent-child) relationship, it has the same spilling effects as the candidate containing the ancestor (parent resp.) node.*

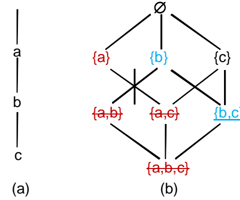


Figure 10. Query Tree and Its Spill Candidates

For instance, in Figure 10(b), the underlined candidate $\{b, c\}$ has the same spilling effect as $\{b\}$. The candidates which have strikethrough have the same spilling effect as $\{a\}$. Clearly, we should avoid generating such candidates with the same spilling effects. Hence now we propose a minimum non-redundant search space.

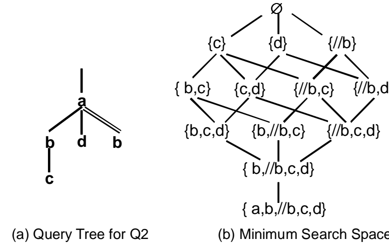
Minimum Search Space. We design an algorithm that generates a minimum set of all non-redundant spill candidates. The idea is to generate non-redundant candidates from the subtrees recursively. For a tree of height h, to generate all possible non-redundant candidates, it picks zero or one candidate from candidates generated by each subtree of height $h - 1$ and composes them to one new candidate. Or it can generate a new candidate which consists of a single root node. The minCandidates algorithm is described below:

Let us assume the height of query tree is h and the width is w. In general, we derive the complexity equal to $2^{w^{h-1}}$. When the tree is deep and narrow (small width and large height), the complexity is low. When the tree is shallow and wide, the complexity is high. The worse case is when the tree has two levels (its width becomes

Algorithm 1 minCandidates

Input: Query Tree T
Output: candidate set S
void minExhaust(Node root)
if root is leaf **then**
  return $\{root\}$;
else
  for each child C_i **do**
    S_i = minCandidates(C_i);
    $S_i = S_i \cup \{\emptyset\}$;
  end for
  //Assume root has w children. Generate candidates.
  $S = S_1 \times S_2 ... \times S_w$;
  $S = S \cup \{root\}$;
  return S;
end if

$|T| - 1$). The complexity is $2^{|T|-2}$. The minimum search space for query Q2 is shown in Figure 11. Note that the size is much smaller than the size of the original search space which is $2^5 = 32$.



(a) Query Tree for Q2 (b) Minimum Search Space

Figure 11. Minimum Search Space for Q1

7.2 Formulation of Optimization Problem

For each spill structure candidate, a reduced query is derived to produce reduced output. Our goal is to pick optimal structures to spill so to optimize output quality. The problem can be formulated as follows:

Given the following inputs: 1. Data arrival rate λ in the number of topmost elements per time unit; 2. Unit qualities gained by executing each reduced query on a topmost element $\{\nu_0, \nu_1, ..\nu_n\}$; 3. Unit processing cost for each candidate reduced queries $\{C_0, C_1, ..C_n\}$. 4. The number of time units for a reduced query to execute, C, denoting the available CPU resources. We aim to find a spill candidate whose corresponding reduced query satisfies the two conditions: (1) Consume all input elements in C time units; (2) Maximize total output utility.

Given a spill candidate, we first derive its corresponding reduced query Q_i. We then use C/C_i to calculate how many elements can be processed when executing Q_i. However, since the processed data cannot exceed the incoming data, the total output quality can be calculated using the formula below:

$$\nu_i * min\{\lambda * C, C/C_i\} \quad (8)$$

7.3 Algorithms for Spill Optimization

Optimal Reduction(OptR). Optimal Reduction approach (OptR) is an exhaustive approach. It searches the entire solution space and picks the candidate which yields the highest output quality.

The procedure is as follows: 1) Iterate every spill candidate in a top-down manner in the spill candidate lattice and derive a reduced query Q_i. 2) Then we estimate the cost, unit quality as well as total output quality of the new reduced query. The candidate query that has the highest output quality will be chosen as the reduced query at spilling stage.

Example 7.1 *Assume the arrival rate is 84 topmost elements/s, i.e. 0.084 elements/ms. Assume the unit cost and unit quality for initial query are 24ms and 6 respectively. The available CPU resources are 240 ms. In this case, the reduced query needs to process 20 topmost elements while achieving the highest output quality. Each candidate is annotated with their unit processing costs and unit quality in Figure 12(a). We finally pick spill candidate* $\{b, c\}$ *since its corresponding reduced query yields the highest final output quality which is (240/12)*2 =40.*

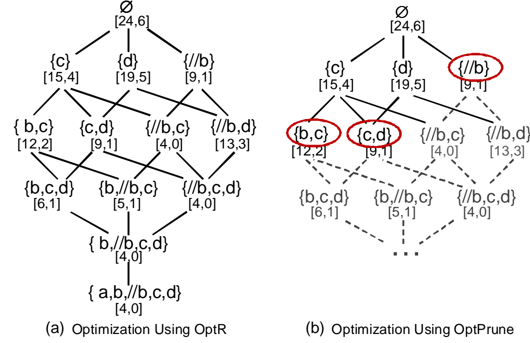


Figure 12. Optimization Using OptR and OptPrune

Optimal Reduction with Pruning (OptPrune). Optimal Reduction with Pruning (OptPrune) applies additional pruning rule to eliminate suboptimal solutions. It again explores the search space in a top-down manner and removes less promising solutions based on the observation below:

Observation 7.2 *In the search space when we reach a candidate* d_i *and find it is capable to consume all input data. The candidates below it (candidates which include all paths in* d_i*) can all be pruned.*

The reason is that if a candidate d_i can produce r_i result structures, the candidates below it tend to spill more paths. In this case the quality of each result structure is definitely no better than that of candidate d_i. However, the number of output result structures keeps the same since all input data is consumed. In this sense, the total output quality of the candidate below d_i is guaranteed not to be higher than that of d_i.

Example 7.2 *In Figure 12(b), candidate* $\{b, c\}$ *can consume all input. In this case, we can prune candidates below it, (*$\{b, c, d\}$*,* $\{b, //b, c, d\}$ *and* $\{a, b, //b, c, d\}$ *directly. Similarly, all candidates below the other circled candidates in Figure 12(b) can be removed.*

Top-down Expansion Heuristics (ToX). We now present a Top-down eXpansion heuristics (ToX) whose complexity is much cheaper than both OptR and OptPrune. ToX heuristics start from simple spill candidates first and stop at the first candidate which is able to consume all the input.

ToX proceeds as follows:

Step 1. Check candidates which spill only one leaf node (candidates on the top level of the lattice). If we can find a candidate which is able to consume all input and achieve highest total output quality among candidates considered so far, stop. Otherwise go to step 2.

Step 2. Pick the candidate which has the highest quality/cost ratio on this level and move to candidates connecting it one level lower.

Step 3. If one of the new candidates can consume all the input and achieve the highest total output quality among candidates considered so far, stop. Otherwise go back to step 2.

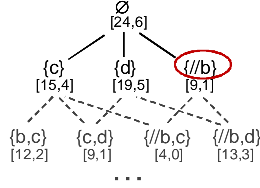


Figure 13. Optimization Using ToX

Example 7.3 *For the search space in Figure 13, we first check the candidates which only spill one node. We find* $\{//b\}$ *can consume all input. We consider* $\{//b\}$ *optimal and stop. The total output quality in this case is 20*1 = 20. ToX is not guaranteed to find an optimal solution.*

8 Experimental Results

In this section, we conduct a comparative study of the three optimization algorithms OptR, OptPrune and ToX. To test whether the selection of substructures is effective, we also employ a greedy selection of substructures. The algorithm, called *Random*, iteratively selects one among all possibly substructures randomly until enough substructures are spilled so that the input load can be handled by the corresponding reduced query. The experimental results demonstrate that our proposed solutions consistently achieve higher quality compared to Random approach.

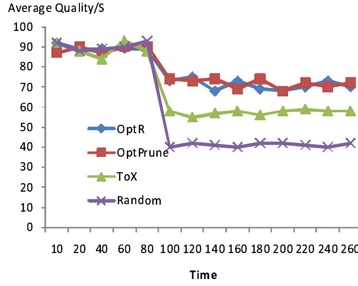
8.1 Experimental Setup

We have implemented our proposed approaches in an XML stream system prototype called Raindrop [12]. We use both real data set and synthetic data set. Real data set is from Mondial, a world geographic database [20]. The testing data file size is 1MB. Synthetic data is the auction data generated using ToXgene [4]. We generate two sets of data which differ in their element sizes in the "return" paths. One data set has equal sizes of elements in the "return" paths while the other set has rather different sizes of elements in the "return" path. The testing data files are about 30MB. All experiments are run on a 2.8GHz Pentium processor with 512MB memory. We conduct experiments using two settings:

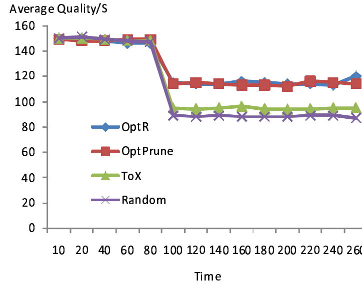
- **Fixed Arrival Rate**. In this case, we use a fixed arrival rate throughout whole query execution. The arrival rate is higher than the processing speed. Spilling is invoked as soon as the memory threshold is reached. Thereafter all methods would continue to spill. We compare the output quality achieved using our proposed approaches with Random approach.
- **Varying Arrival Rate**. In this case, we increase the arrival speed periodically. When the arrival rate increases, the query engine may have to spill more structures to consume all the input. When the arrival rate changes, the spilling algorithm will recalculate and derive the optimal structures to spill. We perform the experiments to study the effect on output quality due to varying arrival rates.

8.2 Fixed Arrival Rate Scenario

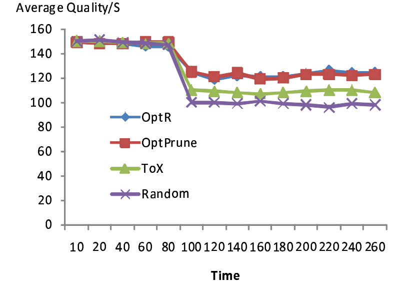
In this set of experiments, we study the output quality gained by taking different optimization approaches. Figures 14(a), 14(b) and 14(c) show the output quality using four optimization strategies on different data sets. At the early stage since the memory threshold is not yet reached, the initial query is running in all cases whose output quality is the highest. After the memory threshold is reached, the spilling stage is invoked. The output quality is decreased since all four approaches spill some structures. However, our proposed approaches, OptR, OptPrune



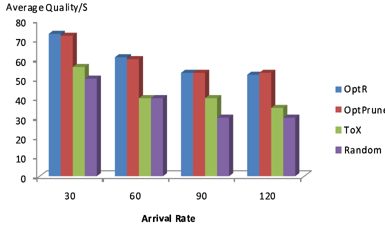
(a) Mondial, fixed arrival rate



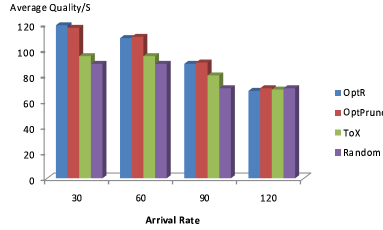
(b) Auction, fixed arrival rate, same size



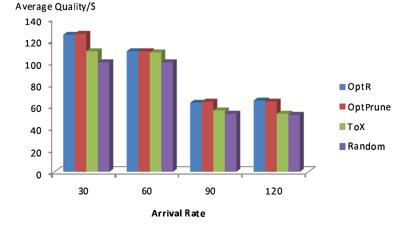
(c) Auction, fixed arrival rate, diff. Size



(d) Mondial, diff. arrival rate



(e) Auction, diff. arrival rate, same size



(f) Auction, diff. arrival rate, diff. size

Figure 14. Performance Comparison of Four Approaches

and ToX gain higher quality than Random. Note that OptR and OptPrune both gain much higher quality than Random and ToX. This is because OptR and OptPrune guarantee to find the optimal structures to spill.

Observe that when the element sizes of the "return" paths are different in Figure 14(c), the output quality is different from the quality in the case of the same size. This is because the costs for each candidate reduced queries are changed when spilling varying structures. Our optimization algorithms will pick different optimal spill structures which yield best quality results.

8.3 Varying Arrival Rate Scenario

Figures 14(d), 14(e) and 14(f) compare the output quality when the arrival rate changes. Note that when the arrival rate is increased, the quality is decreased or keeps the same. This indicates that due to increased arrival rates, the reduced query may need to spill more structures to consume all the input. We also observe that OptR, OptPrune always gain much higher quality than ToX and Random even when the quality is decreased. In other words, our optimization algorithms will pick different optimal spill structures to achieve best quality results for each of the different arrival rate cases.

8.4 Overhead of Spilling Approaches

Here we study the overhead of three spilling strategies. The overhead is measured by the time spent on choosing which structure to spill. We study the relationship between the complexity of the query and the overhead of optimization methods. We use five queries which vary in the sizes of query trees. In Figure 15, when the queries become complex, the overhead of ToX is always smaller than OptPrune and OptR since it stops at the earliest candidate which consumes all input. Observe that the overhead of OptPrune is much smaller than that of OptR. This indicates that our pruning method is indeed effective at reducing the searching cost. Given that both approaches can achieve the highest quality, OptPrune is obviously a better option than OptR. In addition, the overhead of OptPrune does not scale although its overhead is higher than that of ToX. Therefore OptPrune is better than other two approaches since it yields the highest quality results with acceptable overhead.

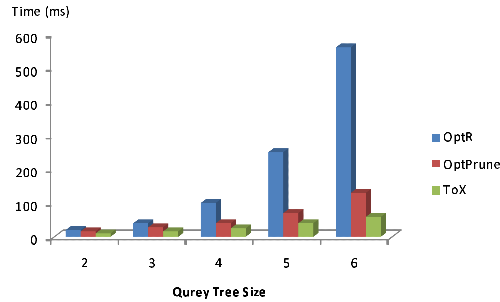


Figure 15. Overhead of Three Approaches

9 Related Work

Spilling techniques have been investigated in relational streams. Flush algorithms have been proposed to either maximize the output rate or to generate a subset of result set as early as possible [16, 23, 27, 28]. However, we cannot directly apply their techniques into structure-based spilling in XML streams because of the following two reasons: 1) The spilled objects in relational streams are tuples. However, in our context, spilled objects are substructures of the hierarchical XML data. 2) These works are focusing on providing non-blocking flush techniques when conducting different relational join, such as Symmetric Hash Join, Hash-Merge Join and Progressive Merge Join. However, structural join is the core component of XQuery plans, which can be looked as a θ join whose condition is to compare the regions of two elements [30].

[25] first proposed to produce approximate results for XQuery when no input for some operators in the plan exists. However, they do not address the case that substructures are missing from the input. In addition, since they assume the data is persistent, supplementary query result generation does not require spilling extra data. Query relaxation techniques are proposed to return approximate query results [2, 15]. However, query relaxation often returns a superset of the results set. They thus target a different objective from ours.

10 Conclusion

We propose a new structure-based spilling strategy that exploits features specific to XML stream processing. We design an H^+ algorithm which can determine containment relationships between two paths efficiently. A complementary output model is proposed to help supplementary results to match reduced output. We examine how to generate reduced output as well as supplementary results caused by spilling. We develop three optimization strategies, OptR, OptPrune and ToX. The experimental results demonstrate that our proposed solutions consistently achieve higher quality results compared to state-of-the-art techniques.

References

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient xml query pattern matching. In *IEEE International Conference on Data Engineering (ICDE)*, page 141, Feb 2002.

[2] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree pattern relaxation. In *EDBT*, pages 496–513, London, UK, 2002. Springer-Verlag.

[3] B. Babcock, M. Datar, and R. Motwani. Load shedding techniques for data stream systems. In *MPDS*, 2003.

[4] D. Barbosa, A. O. Mendelzon, J. Keenleyside, and K. A. Lyons. Toxgene: An extensible template-based data generator for xml. In *WebDB*, pages 49–54, 2002.

[5] C. Koch et al. FluxQuery: An Optimizing XQuery Processor for Streaming XML Data. In *VLDB*, pages 228–239, 2004.

[6] L. Chen. *Semantic Caching for XML Queries*. PhD thesis, Worcester Polytechnic Institute, 2004.

[7] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The next framework for logical xquery optimization. In *In VLDB*, pages 168–179, 2004.

[8] Y. Diao and M. Franklin. Query Processing for High-Volume XML Message Brokering. In *VLDB*, pages 261–272, 2003.

[9] L. Fegaras. The Joy of SAX. In *First International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*, 2004.

[10] L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi. Query processing of streamed xml data. In *CIKM*, pages 126 – 133, 2002.

[11] A. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *ACM SIGMOD*, pages 419–430, 2003.

[12] H. Su, J. Jian and E. A. Rundensteiner. Raindrop: A Uniform and Layered Algebraic Framework for XQueries on XML Streams. In *CIKM*, pages 279–286, 2003.

[13] G. Häubl and V. Trifts. Consumer decision making in online shopping environments: The effects of interactive decision aids. *Marketing Science*, 19(1):4–21, 2000.

[14] Z. Ives, A. Halevy, and D. Weld. An XML Query Engine for Network-Bound Data. *VLDB Journal*.

[15] Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. In *PODS*, pages 40–51, New York, NY, USA, 2001. ACM.

[16] R. Lawrence. Early hash join: a configurable algorithm for the efficient and early production of join results. In *VLDB*, pages 841–852, 2005.

[17] B. Ludascher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *VLDB*, pages 227–238, 2002.

[18] M. F. Fernandez, D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *ICDE*, pages 14–23, 1998.

[19] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries on Heterogeneous Data Sources. In *Proceedings of the 27th VLDB Conference, Edinburgh, Scotland*, pages 241–250, 2001.

[20] W. May. Information extraction and integration with florid: The mondial case study, 1999.

[21] G. Miklau and D. Suciu. Containment and equivalence for an xpath fragment. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*.

[22] G. Miklau and D. Suciu. Containment and equivalence for a fragment of xpath. *J. ACM*, 51(1):2–45, 2004.

[23] M. Mokbel, M. Lu, and W. Aref. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *ICDE*, page 251, 2004.

[24] F. Peng and S. Chawathe. XPath Queries on Streaming Data. In *ACM SIGMOD*, pages 431–442, 2003.

[25] J. Shanmugasundaram, K. Tufte, D. J. DeWitt, J. F. Naughton, and D. Maier. Architecting a network query engine for producing partial results. In *WebDB*, pages 17–22, 2000.

[26] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.

[27] W. H. Tok, S. Bressan, and M.-L. Lee. A stratified approach to progressive approximate joins. In *EDBT '08: Proceedings of the 11th international conference on Extending database technology*, pages 582–593, New York, NY, USA, 2008. ACM.

[28] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.

[29] Y. Wu, J. M. Patel and H. V. Jagadish. Structural Join Order Selection for XML Query Optimization. In *ICDE*, pages 443–454, 2003.

[30] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 30(2):425–436, June 2001.