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

Mingzhu Wei, Elke A. Rundensteiner
Worcester Polytechnic Institute, USA
samanwei|rundenst@cs.wpi.edu

Murali Mani
University of Michigan, Flint
mmani@umflint.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 engines. In this work, we now introduce "structure-based spilling", a spilling technique customized for XML streams by considering the partial spillage of possibly complex XML elements. Such structure-based spilling brings new challenges. When a path is spilled, multiple paths may be affected. We analyze possible spilling effects on the query paths and how to execute the "reduced" query to produce partial results. To select the reduced query that maximizes output quality, we develop three optimization strategies, namely, OptR, OptPrune and ToX. We also examine the clean-up stage to guarantee that an entire result set is eventually generated by producing supplementary results. Our experimental study demonstrates that our proposed solutions consistently achieve higher quality results compared to the state-of-the-art techniques.

1. INTRODUCTION

Motivation. XML stream systems have attracted researchers' interest recently [1–6] because of the wide range of potential applications such as publish/subscribe systems, supply chain management, financial analysis and network intrusion detection. Different from relational stream systems, XML stream processing experiences new challenges: 1) the incoming data is entering the system at the granularity of a continuous stream of tokens, instead of self-contained tuples. This means the engine has to extract relevant tokens to form XML elements. 2) We need to conduct dissection, restructuring, and assembly of complex nested XML elements specified by query expressions, such as XQuery.

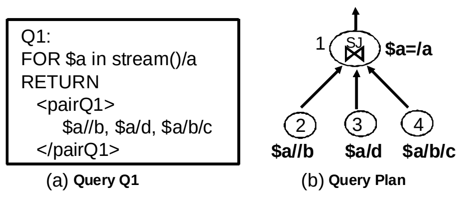
For most stream applications, immediate online results are required, yet network traffic may be unpredictable. When the arrival rate is high, stream processing systems may not be able to keep up with the input data - resulting in buffer overflow or uncontrolled data loss. To produce eventually complete results, load spilling, which pushes some fractions of data to disks temporarily, is employed in relational stream engines [7–10]. 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 result structures is 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. We now motivate the practicability of structure-based spilling via concrete application scenarios below.

Example 1. In online auction environments, sellers may continuously start new auctions. When customers 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 displayed first, which aid customers in making decisions. Many consumers tend to use a two-stage process to reach their decisions [11] instead of inspecting 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 features, are often evaluated later after consumers have identified their favorite subsets. When system resources are limited, the query engine may spill unimportant attributes to 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 discarded packets may contain the information related to the attack. In this case, dropping 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.

Let us look at a structural spilling example. Query Q1 and its



```
Q1:
FOR $a in stream()/a
RETURN
  <pairQ1>
     $a//b, $a/d, $a/b/c
  </pairQ1>
```

Figure 1: Query Q1 and Its Plan

plan are shown in Figure 1. Query Q1 returns three path expressions, $\$a//b$, $\$a/d$ and $\$a/b/c$. The plan conducts structural joins on the binding variable $\$a$ and these three path expressions. In this work, we assume any path and any number of paths in the query can be spilled to disk when the 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 data corresponding to paths 2 and 4 in the plan is actually affected (as side effect) by such spilling. For each output tuple (e.g., <pairQ1> in Q1), partial result structures are produced since both b and c elements are missing. In this case, several savings arise. First, since complete b elements are pushed to disk from the token stream, we do not need to bother to extract "c" elements from the input at this time. In other words, we bypass the processing of tokens from "<c>" to "</c>". Second, we no longer need to conduct structural joins between $\$a$ and $\$a//b$ nor between $\$a$ and $\$a/b/c$. Henceforth, we refer to the user query after spilling has been applied as *reduced query* and the early output produced by it as *reduced output*.

Such structural-based spilling brings new challenges which do not exist in relational streams. There are many options to spill paths from a given query. Different reduced queries may vary in their processing costs and output quality. Hence the correct choice of appropriate reduced query raises many issues: 1) which additional paths in the query are affected by spilling a particular path; 2) how to estimate the cost of alternative reduced queries as well as the partial result quality; and 3) which potential reduced query should be chosen to obtain maximum output quality. We tackle these challenges using a three-pronged strategy. One, we examine how to execute reduced queries given varying spilling effects on the query. Two, we provide metrics for measuring the quality and cost of the alternative reduced queries. Three, we transform the reduced query selection problem into an 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.

In addition, to eventually produce entire yet duplicate-free result set, we need to generate supplementary results correctly at a later time when the system has sufficient computing resources. For this, we design an output model to match supplementary "delta" structures with partial result structures produced earlier. To generate supplementary results, we determine what extra data to flush to disk to guarantee that the entire result set can be produced.

Contributions. Our contributions are summarized as below:

1. We propose a general framework to address structure-based spilling which can be applied in any XML stream system.

2. We formulate our structure-based spilling problem into an optimization problem, namely, to find the reduced query that maximizes the output quality based on our structure-based quality and cost model for XML streams.

3. We study the effect on different paths in the query for a particular spilled path and examine how to execute the reduced query.

4. To solve the spilling problem, we develop a family of three optimization strategies, OptR, OptPrune and ToX. OptR and OptPrune are both guaranteed to identify an optimal reduced query, with OptPrune exhibiting significantly less overhead than OptR. ToX uses a heuristic-based approach, which is much more efficient than OptR and OptPrune.

5. We propose an output quality model for evaluating the output quality for different reduced queries, and a cost model for evaluating the execution cost for different reduced queries. This output quality model and the cost model are used by our algorithms.

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

2. OVERVIEW OF OUR APPROACH

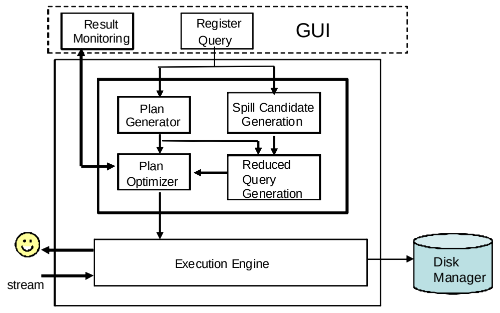


Figure 2: Architecture for Spilling Framework

The architecture of our spilling framework is shown in Figure 2. After the queries are registered with the query engine, an initial plan is generated and optimized. The execution engine will instantiate the query plan and start processing input streams. The problem of deciding when the system needs to spill data is not a question specific to XML stream. Any existing approach from the literature [7, 8] could be employed here. We employ a memory buffer to store input stream data. As soon as a token is processed, we clean this token from the buffer. We assume a threshold on the memory buffer that allows us to endure periodic spikes of the input. When buffer occupancy exceeds the given threshold, we trigger the spilling.

When spilling is triggered, first, the possible spilling candidates are examined. We then derive the reduced queries for each spilling candidate. The query optimizer runs the optimization algorithm to pick the optimal reduced query. Finally the reduced query is instantiated, in place of the previously active query, initiating the spilling process. Later when the arrival speed becomes near zero, we invoke the clean up processing to generate supplementary results based on disk-resident data.

Recall that any path and any number of paths in the query can be spilled. We describe the details of possible spilling candidates in Section 4. Now let us illustrate how to pick the optimal spilling candidate to produce maximum output quality. We require the optimal reduced query should be able to consume all the input, i.e., the processing speed of the optimal reduced query should be faster than or equal to the arrival rate. For example, assume we have two spilling candidates for Q1, $/a//b$ and $/a/b/c$. The data is shown in Figure 3(a). Figures 3(b) and (c) list output results after spilling $/a//b$ and $/a/b/c$ respectively. Assume the arrival rate is 500 topmost elements/sec (for Q1, a is the topmost element). Assume the cost to produce each <pairQ1> element when spilling $/a//b$ is 0.6 milliseconds. The cost of producing each <pairQ1> when spilling $/a/b/c$ is 1 millisecond. The processing rates when spilling $/a//b$ and $/a/b/c$ are 1000/0.6 =1333 and 1000/1=1000 respectively. Both values are greater than the arrival rate. Therefore spilling either $/a//b$ or $/a/b/c$ can both meet our goal of consuming all the input. However, the output quality for each spilling path is different. When spilling $/a//b$, since only d elements are present in the results, the quality for each <pairQ1> is 1 (quality computation is detailed in Appendix D). The quality when spilling $/a/b/c$ is 3 since b (including partial b and complete b) and d elements are returned. In this case, the output quality when spilling $/a/b/c$ within 1 second is 500*3 and the quality when spilling $/a//b$ is 500*1. Therefore spilling path $/a/b/c$ yields higher output quality than $/a//b$. We will describe the detailed algorithm to find an optimal candidate in Section 6. This structural spilling framework is general and can be applied in any XML stream engine. The detailed explanation of why our spilling framework is general can be found in Appendix B.

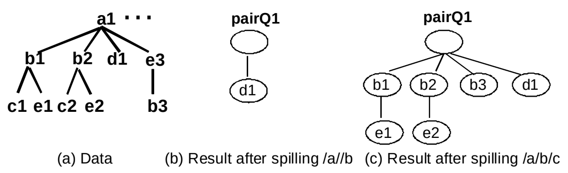

a1 · · ·
b1
b2
d1
e3
c1
e1
c2
e2
b3
pairQ1
d1
pairQ1
b1
b2
b3
d1
e1
e2
(a) Data
(b) Result after spilling /a//b
(c) Result after spilling /a/b/c

Figure 3: Data and Output for Q1

To eventually produce the entire yet duplicate-free result set, we have to generate supplementary results correctly. We propose a complementary output model, which extends from the hole-filler model in [12], to facilitate the matching of the supplementary results with prior generated output. In addition, we examine what extra data must be flushed to guarantee the generation of the correct "delta" structure in supplementary results. The details of generating supplementary results can be found in Appendix E.

3. BACKGROUND

Queries Supported. We support a subset of XQuery in this work. Basically, we allow (1) "for... where... return..." expressions (referred to as FWR) where the "return" clause can further contain FWR expressions; and (2) conjunctive selection predicates where each predicate is an operation between a variable and a constant. The grammar of the supported XQueries can be found in Appendix A. A large range of common XQueries can be rewritten into this subset [13]. The rewriting rules for some forms of queries, such as queries with "let" clause, or queries with FWR expressions nested within "for" clause, can be found in Appendix A.

Algebraic Query Processing. We assume the queries have been normalized using the techniques in [14]. Queries are then translated into a plan. Namely, 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 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, 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 example, for query Q1, 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 [15]. In this work, BPU is the topmost element in the query tree.

4. SPILL CANDIDATE SPACE

In this section we examine all possible spill candidates. To do this, we represent the query using a query pattern tree. For example, the query pattern tree for Q1 is given in Figure 5(a). Each node in the query tree indicates an XPath expression. The semantics of the supported XPath expression can be found in Appendix A. We use single line edges to denote the parent-children relationship and double line edges to denote the ancestor-descendant relationship.

We assume any node and any number of nodes in the query tree can be spilled. Each of them forms a spill candidate. To analyze the total number of potential spilling candidates, consider a complete query pattern tree with depth d and fixed fan-out f. The total number of nodes in the query tree $|T| = \sum_{i=1}^{d-1} f^i = \frac{f^d - 1}{f-1}$. Since any number of nodes in the query tree can be spilled, the total number of potential spilling candidates is $C^0_{|T|} + C^1_{|T|} + ... + C^{|T|}_{|T|} = 2^{|T|}$, which is bounded by $O(2^{f^d})$.

An example query tree and its possible candidates are shown in Figure 4. Query tree is shown on the left and its possible candidates are shown on the right. Each node in the lattice represents one candidate. The top candidate 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. The candidate space scales quickly since it is exponential in the number of nodes in the query tree.

We now reduce the spill candidate space using the insight that some candidates may result in the same spilling effects. Recall that when we spill data corresponding to a path p from the query tree, all its descendants are also flushed to disk. This leads to the following observation:

OBSERVATION 4.1. *If a spill candidate includes two nodes which satisfy the ancestor-descendant (or parent-child) relationship, it has the same spilling effect as the candidate containing the ancestor (parent resp.) node.*

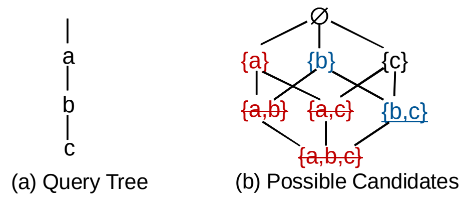


Figure 4: Query Tree and Its Spill Candidates

For instance, in Figure 4(b), the underlined candidate $\{b, c\}$ has the same spilling effect as $\{b\}$. The candidates with strike-through have the same spilling effect as $\{a\}$. Clearly, we should avoid examining such candidates with the same spilling effects. Hence we introduce a minimum non-redundant spill candidate space.

Minimum Candidate Space. We design an algorithm that generates the 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 the set of candidates generated by each subtree of height $h-1$ and composes them to one new candidate. Or, it can also generate a new candidate which consists of a single root node. The detailed algorithm is described in Appendix C. The total number of potential spilling candidates generated using this algorithm is $O(2^{fd})$. The minimum spill candidate space for query Q1 is shown in Figure 5(b). Its size is much smaller than that of the original candidate space which is 2^5 = 32.

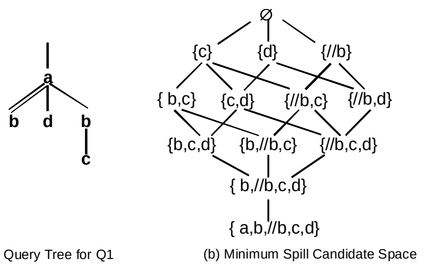


Figure 5: Minimum Candidate Space for Q1

5. GENERATE CORRECT REDUCED OUTPUT

5.1 Determine Spilling Effects

For each spill candidate, we need to derive its corresponding reduced query and generate the correct reduced output. As shown in Section 1, when a path is spilled, multiple paths in the query may be affected. To generate the reduced output correctly, we have to determine the spilling effects on the paths in "for", "where" and "return" clauses for each spilling candidate. Each path in the query corresponds to a set of subtrees in the document. For instance, $/a/b$ returns the subtrees rooted at nodes b whose parents are of type a. Due to spilling, either the root or the non-root nodes in the subtree can be missing. Here we define two categories of spilling effects on paths in the query to distinguish between different missing locations of the subtrees:

- **Root missing or unaffected**. When the roots of subtrees for a query path are missing, we call this *root missing*. Otherwise, it is *unaffected*. For instance, for path $/a//b$, the roots of some subtrees are missing when spilling $/a/b$. This is because path $/a/b$ is contained by $/a//b$. In other words, they satisfy the following relationship:

$$P \bigcap S//* \neq \emptyset \tag{1}$$

  Here P indicates a path in the query and S indicates the spilled path.

- **Subpart missing or unaffected**. When non-root nodes in the subtrees corresponding to a path in the query are missing, we call it *subpart missing*. Otherwise, it is unaffected. For instance, $/a/b$ is subpart missing when spilling $/a/b/c$ because c nodes in the subtrees are missing due to spilling. The query paths which are subpart missing satisfy the following relationship:

$$P/*//* \bigcap S//* \neq \emptyset \tag{2}$$

To determine root missing and subpart missing, we use the approach in [16] which constructs the product automaton of P and S. The complexity of this approach is O(|P|*|S|). Since these two categories are orthogonal, there are 2*2=4 combinations. They are:

- *Root missing and subpart missing* (SRAM). E.g., when spilling $/a//b$, $/a/b$ is SRAM because both root and subpart are missing.
- *Root unaffected and subpart missing* (SAM). E.g., $/a/b$ is SAM when spilling $/a/b/c$ since c nodes in subtrees are missing.
- *Root missing and subpart unaffected* (RAM). This is not possible. Because we assume when a path is spilled, all its descendants are also spilled.
- *Root unaffected and subpart unaffected* (UA). In this case, both root and subpart are unaffected.

5.2 Reduced Query Execution

We now describe how to execute a reduced query based on the knowledge of spilling effects. The reduced query results are output as long as the result is correct, even if the result structures are partial. In other words, the reduced query execution should satisfy the maximal output property [17]. Therefore we propose the following policies for reduced query execution so that we can produce as much correct output as possible.

- **Affected path in "for" clause**. When the binding variable is SRAM, the number of bindings may be reduced. In this case we can still produce output as long as the binding variable does not return empty. When the binding variable is subpart missing (SAM), although a subpart of the binding variable is missing, it does not affect the number of iterations of the "loop counter". Therefore SAM on the "for" path does not affect result generation.

  EXAMPLE 5.1. *Figure 6(a) shows the case when the binding variable is SAM. In Figure 6(a), the spilled path is $/a/b$. The binding variable $\$a$ is SAM due to spilling $/a/b$. The iterations of "for" loop are unaffected.*

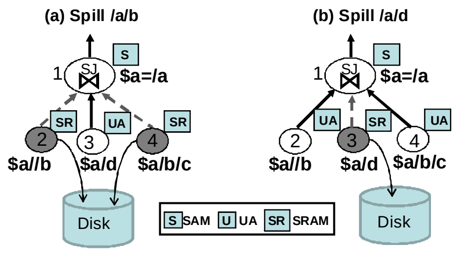


Figure 6: Plan for Q1 with Spilling Effects

- **Affected path in "return" clause**. 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 FWR block.

  EXAMPLE 5.2. *Figure 6(a) shows the case that the returned paths $\$a//b$ and $\$a/b/c$ are both SRAM due to spilling $/a/b$. For data in Figure 3(a), only $b3$ and $d1$ are present in the $< pairQ1 >$ results. In Figure 6(b), $/a/d$ is spilled. Only $\$a//b$ and $\$a/b/c$ produce results. In both cases, returned pairQ1 elements are partial since they are not composed of all the returned substructures.*

- **Affected path in "where" clause**. When a "where" path falls into SAM, if the missing subpart is not needed for the predicate evaluation, we do not block the predicate evaluation. However, when the "where" path is SRAM, the predicate evaluation cannot be conducted on all the elements. In this case, we may not know whether the results should be output or not. Therefore we treat affected SRAM on the "where" paths as blocking. Whenever a "where" path is SRAM, the output for its corresponding FWR and its inner FWR block thus do not produce anything in our model.

```
Q2: FOR $a in stream()/a
    WHERE $a/d="55"
    RETURN <pairQ2>
        $a/d/f, $a/e, $a/b/c
              </pairQ2>
```

```
Q3: FOR $a in stream()/a
    RETURN <result>$a/c,
          FOR $b in $a/b
          WHERE $b/e ="6"
          RETURN $b/f
             </result>
```

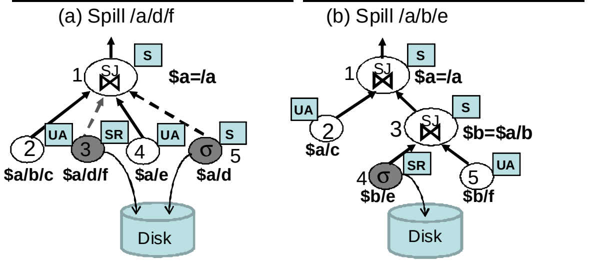


Figure 7: Reduced Query Plans for Q2 and Q3

EXAMPLE 5.3. *Query Q2 has a predicate on $\$a/d$. Figure 7(a) shows the reduced query plan when spilling $/a/d/f$. "Where" path $\$a/d$ is SAM. In this case, the predicate evaluation is not affected and we can return partial results. Now let us look at Q3 which has a predicate in the inner FWR block. Figure 7(b) shows the reduced plan when spilling $/a/b/e$. For the inner FWR block, since $\$b/e$ is SRAM, the predicate evaluation cannot be conducted. Therefore the inner FWR block cannot produce $\$b/f$. However, since $\$a/c$ in the outer FWR block is unaffected, we can produce $\$a/c$ in the result.*

6. CHOOSE THE OPTIMAL STRUCTURE TO SPILL

6.1 Formulation of Optimization Problem

For each spill candidate, a reduced query is derived to produce the reduced output. For each reduced query, we measure its unit quality and unit processing cost. Unit quality for a reduced query is defined as the quality gained by executing the reduced query on a topmost element. Unit processing cost is the average time of processing a topmost element. The detailed description of our quality and cost model can be found in Appendix D. Our goal is to pick structures to spill so as to optimize the 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 quality gained by executing each reduced query $\{\nu_0, \nu_1, ..\nu_n\}$; 3. Unit processing costs for each candidate reduced query $\{C_0, C_1, ..C_n\}$. We aim to find a spill candidate whose corresponding reduced query satisfies the following two conditions: (1) Consume all input elements in 1 time unit; and (2) Maximize total output quality.

Given a spill candidate, we first derive its corresponding reduced query Q_i. We use $1/C_i$ to calculate how many elements can be processed when executing Q_i per time unit. Since the processed data cannot exceed the incoming data, the total output quality is calculated using the formula below:

$$\nu_i * min\{\lambda, 1/C_i\} \tag{3}$$

6.2 Algorithms for Spill Optimization

Optimal Reduction(OptR). The first algorithm we propose, called Optimal Reduction (OptR), employs an exhaustive approach. It searches the entire candidate space and picks the candidate which yields the highest output quality.

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

Remember from Section 4 that f is the fan-out and d is the depth of the query pattern tree. Since it is an exhaustive approach, the search complexity is equal to the size of the minimum candidate space, which is $O(2^{fd})$.

EXAMPLE 6.1. *Assume the arrival rate is 20 topmost elements/s. The unit cost and unit quality for the initial query are 0.1s and 6 respectively. The available CPU resources are 1 second. In this case, the reduced query needs to process 20 topmost elements while achieving the highest output quality. The unit processing cost and unit quality for each candidate are shown in Figure 8. We pick spill candidate $\{b, c\}$ since its corresponding reduced query yields the highest output quality, namely, (1/0.05)*2 =40.*

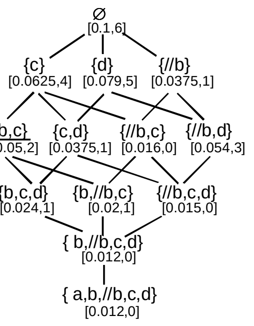


Figure 8: Optimization Using OptR

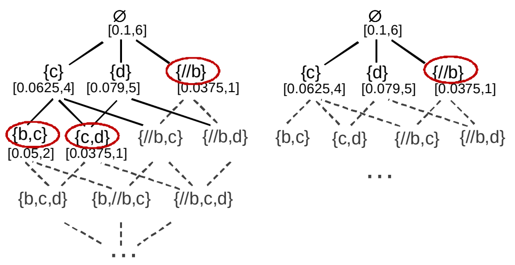
Optimal Reduction with Pruning (OptPrune). Optimal Reduction with Pruning (OptPrune) applies additional pruning to eliminate suboptimal solutions. It explores the spill candidate space in a top-down manner and removes less promising solutions based on the observation below.

OBSERVATION 6.1. *In the top-down candidate space traversal, when we reach a candidate d_i and find it is capable to consume all input data, then the candidates below it (candidates which include all paths in d_i) can all be pruned.*

The reason is that if candidate d_i can produce r_i result structures, the candidates below it tend to spill more paths. The quality of each result structure is not higher than that of candidate d_i. However, the number of output result structures may stay unchanged since all input data is consumed. Therefore, the total quality of the candidate below d_i is guaranteed not to be higher than that of d_i.

EXAMPLE 6.2. *In Figure 9(a), candidate $\{b, c\}$ can consume all input. In this case, we can prune candidates below it, $\{b, c, d\}$, $\{b, //b, c\}$ and $\{b, //b, c, d\}$ directly. Similarly, candidates below $\{//b\}$ and $\{c, d\}$ can be removed.*

To estimate the search complexity, since the worse case for OptPrune is checking every candidate without pruning anything, there-



(a)Optimization Using OptPrune (b)Optimization Using ToX

Figure 9: OptPrune and ToX Example

fore the worst case for OptPrune is $O(2^{fd})$. However, our experimental results will show that the actually complexity is much smaller than $O(2^{fd})$.

Top-down Expansion Heuristic (ToX). We now present a Top-down eXpansion heuristic (ToX), which has much more efficient running time compared to OptR and OptPrune. ToX starts from simple spill candidates and stops at the first candidate which is able to consume all the input.

ToX proceeds as follows:

Step 1. Check candidates which spill one leaf node (candidates on the top level of the lattice). If we 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.

The complexity of ToX is $O(f^{2d})$ which is much smaller than that of OptR and OptPrune.

EXAMPLE 6.3. *In Figure 9(b), 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 is* $min\{$*20, 1/0.0375*$\}$**1 = 20.*

7. EXPERIMENTAL RESULTS

In this section, we conduct a comparative study of the three optimization algorithms OptR, OptPrune and ToX. In addition, we also employ an algorithm, called *Random*, which 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 the Random approach. The experiments are divided into three categories:

- The first set of experiments compares the performance of our proposed spilling strategies with Random approach in two cases. One case is when the network is fast and reliable, i.e, the input sources are never blocked. The other case is when the network is unreliable.
- The second set of experiments tests the impact of different selectivity and different query path sizes on the performance of our approaches.
- The third set of experiments compares the overhead of different spilling approaches.

Experimental Setup. We have implemented our proposed approaches in an XML stream system called Raindrop [15]. The data sets are generated using ToXgene [18]. All experiments are run on a 2.8GHz Pentium processor with 512MB memory.

7.1 Comparison of Spilling Approaches

7.1.1 Reliable networks

A reliable network never incurs suspensions of data transmission. For achieving this, we set arrival interval between two topmost elements to a fixed value. In this set of experiments, we set arrival interval to 0.025s and 0.02s respectively. The arrival rates under these two settings are higher than the processing speed. We use Q1 as the running query. Spilling is invoked as soon as the memory buffer threshold is reached.

To compare the performance of alternative approaches, we use a new "fine-grained" quality metric to measure the quality of partial outputs instead of using traditional throughput metric. The reason is that throughput typically refers to the number of (complete) output elements in XML produced. However, in this work of producing partial structures, a traditional throughput metric is not so meaningful. The detailed quality model can be found in Appendix D.

We study the output quality gained by taking different optimization approaches. Figures 10(a) and 10(b) show the cumulative output quality using four optimization strategies when the arrival interval is 0.025s and 0.02s respectively. Observe that OptR, OptPrune and ToX gain higher quality than Random after spilling starts. OptR and OptPrune both gain higher quality than Random and ToX. This is because OptR and OptPrune guarantee to find the optimal structures to spill.

Because the reliable network never incurs suspension of data transmission, the clean up processing is invoked after all the data has arrived (after time 5500). In the clean up phase, the supplementary results are generated based on the disk-resident data. Finally all four spilling approaches produce the complete result set and reach the same output quality.

When the arrival interval is 0.02s, the cumulative quality increases slower than the case that the interval is 0.025s. This is because when the arrival rate is increased, the reduced query may need to spill more structures to consume all the input.

7.1.2 Unreliable networks

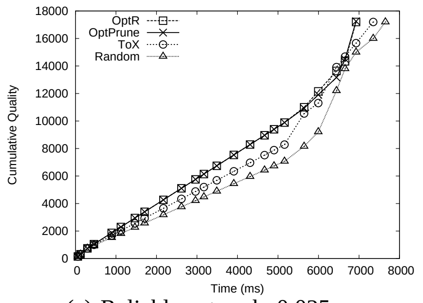
Having evaluated our spilling approaches in the absence of transmissions, we proceed to examine the performance for unreliable networks. To simulate unreliable network, we generate arrival intervals using Pareto distribution that is widely used in case of bursty network [19]. Figure 10(c) shows the cumulative quality for four approaches. Observe that all of them have step-like performance due to switching between the spilling and clean up phase. The slope of segments corresponding to the spilling phase for OptR and OptPrune is larger than that of ToX and Random. This indicates that output quality for OptR and OptPrune is increased faster than that of ToX and Random.

7.2 Impact of Selectivity and Path Size

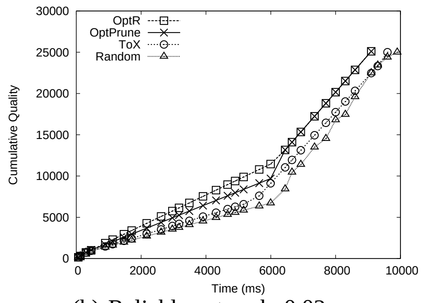
Next, we illustrate that the output quality is affected by the selectivity distribution of the binding variable and each branch. We run the query Q4 below:

```
Q4: FOR $o in stream("test")/list/o
  RETURN $o/P1, $o/P2, $o/P3, $o/P4
```

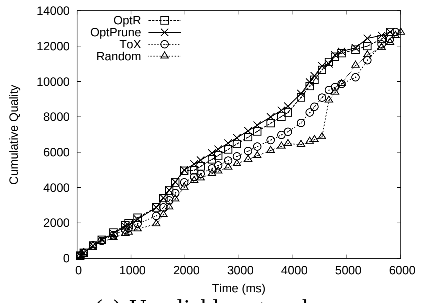
We generate five test data sets which satisfy the following re-



(a) Reliable network, 0.025s



(b) Reliable network, 0.02s



(c) Unreliable network

Figure 10: Performance Comparison of Four Approaches

quirements: 1) all test data sets contain the same number of tokens; and 2) the numbers of elements corresponding to each returned path are equal; and 3) the element sizes corresponding to each returned path are equal. Based on the cost model in Appendix D.2, the locating costs spent on locating each returned path are the same. The join costs between the binding variable and each returned path are the same too. In addition, the spilling costs when spilling each returned path are also the same. For each data set, the selectivity between the binding variable and its branches can be different. We use five different sets of selectivity which differ in their standard deviations. Figure 11 shows that the output quality is higher when there is a bigger variance among selectivity for OptR and OptPrune. This is because OptR and OptPrune tend to spill the return paths with low selectivity which yield low output quality given the same spilling and computation cost. We observe that the quality of the reduced query achieved by the Random approach does not change a lot because Random approach does not keep the returned paths having large selectivity.

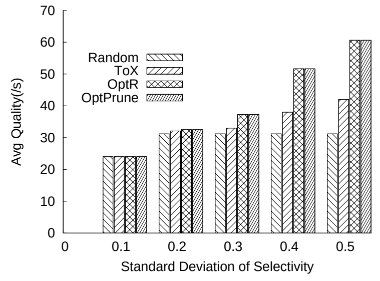


Figure 11: Quality for Varying Selectivity

We now illustrate that the output quality is affected by the pattern size. All testing data sets have the same number of elements and selectivity for each returned paths. And all test data sets contain the same number of tokens. Figure 12 shows that the output quality changes with varying standard deviation of return path size. For the Random approach, the output quality does not change a lot. However, for OptR and OptPrune, the output quality is much higher than the quality achieved by Random approach when the standard deviation of pattern size increases. This is because the reduced queries with smaller returned path size have smaller spilling cost, resulting in lower overall processing cost. In this case, OptR and OptPrune would pick such reduced queries since they have relatively higher quality/cost ratios and thus higher quality.

7.3 Overhead of Spilling Approaches

In this work, optimization is conducted in an online fashion to assure continuous responsiveness of our system. Here we study

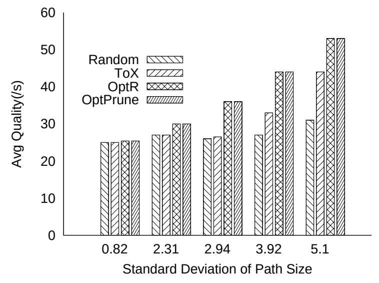


Figure 12: Quality for Varying Path Size

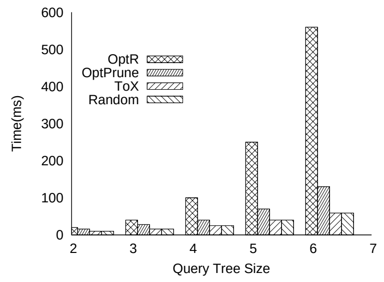


Figure 13: Overhead of Four Approaches

the overhead of four spilling strategies, 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 the optimization methods. We use five queries which vary in the size of the query trees. In Figure 13, when the queries become complex, the overhead of ToX is much smaller than OptPrune and OptR since it stops at the earliest candidate which consumes all input. We 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 search cost. Given that both approaches can achieve the highest quality, OptPrune is obviously a better option than OptR. However, when the query becomes more complex, OptPrune may not be a practical solution since its overhead is larger than ToX and Random. In this case, we resolve to utilize our lightweight ToX solution.

8. RELATED WORK

In relational streams, flush algorithms have been proposed to maximize the output rate or to generate a subset of results as early as possible [7–10]. We can apply their techniques on coarse-grained spilling in XML, which is spilling complete topmost elements to disk. However, such coarse-grained spilling misses the novel XML-specific opportunities for spilling. In this work, we instead focus on the fine-grained XML-specific structural spilling approach.

[17] first proposes 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. [20] addresses structural shedding problem in XML streams. However, it only considers queries containing independent returned paths. Also, since it is focusing on shedding, how to generate supplementary results is not discussed.

[1–6] evaluate XQuery expressions over XML streaming data. One approach [2, 5] combines automaton and algebra to process XML stream data. E.g., Tukwila [5] and YFilter [2] model the whole automaton processing as one mega operator while modeling the rest data manipulation such as filtering and restructuring in algebraic operators. [1, 3, 4, 6] use automata or automaton-like SAX event handlers to process the whole query. As discussed in Appendix B, the only limitation of our structural spilling framework is that the cost model measuring processing costs is related to the specifics of the implementation of query processing. Therefore, we can apply our spilling techniques to other XML stream systems as long as we plug in their cost models.

9. CONCLUSIONS

We propose the first structure-based spilling strategy that exploits features specific to XML stream processing. Our structure-based spilling framework is general and can be applied in any XML stream system. We analyze the effect on different paths in query for a particular spilled path. We design an output quality model for evaluating the quality of partial returned structures. A complementary output model is proposed to match supplementary results with reduced output. To solve the spilling problem, we develop three strategies, OptR, OptPrune and ToX. The experimental results demonstrate that our proposed solutions achieve higher quality results compared to state-of-the-art techniques.

10. REFERENCES

[1] C. Koch, S. Scherzinger, N. Scheweikardt and B. Stegmaier, "FluxQuery: An Optimizing XQuery Processor for Streaming XML Data," in *International Conference on Very Large Data Bases (VLDB)*, 2004, pp. 228–239.

[2] Y. Diao *et al.*, "Query Processing for High-Volume XML Message Brokering," in *International Conference on Very Large Data Bases (VLDB)*, 2003, pp. 261–272.

[3] A. Gupta *et al.*, "Stream Processing of XPath Queries with Predicates," in *ACM SIGMOD*, 2003, pp. 419–430.

[4] B. Ludascher, *et al.*, "A Transducer-Based XML Query Processor," in *International Conference on Very Large Data Bases (VLDB)*, 2002, pp. 227–238.

[5] Z. Ives, *et al.*, "An XML Query Engine for Network-Bound Data," *VLDB Journal*.

[6] F. Peng *et al.*, "XPath Queries on Streaming Data," in *ACM SIGMOD*, 2003, pp. 431–442.

[7] T. Urhan *et al.*, "Xjoin: A reactively-scheduled pipelined join operator," *IEEE Data Engineering Bulletin*, vol. 23, no. 2, pp. 27–33, 2000.

[8] M. Mokbel, *et al.*, "Hash-merge join: A non-blocking join algorithm for producing fast and early join results," in *Proceedings of ICDE*, 2004, p. 251.

[9] R. Lawrence, "Early hash join: a configurable algorithm for the efficient and early production of join results," in *VLDB*, 2005, pp. 841–852.

[10] W. H. Tok, *et al.*, "A stratified approach to progressive approximate joins," in *EDBT '08: Proceedings of the 11th international conference on Extending database technology*. New York, NY, USA: ACM, 2008, pp. 582–593.

[11] G. Häubl *et al.*, "Consumer decision making in online shopping environments: The effects of interactive decision aids," *Marketing Science*, vol. 19, no. 1, pp. 4–21, 2000.

[12] L. Fegaras, *et al.*, "Query processing of streamed xml data," in *CIKM*, 2002, pp. 126 – 133.

[13] I. Manolescu, *et al.*, "Answering XML Queries on Heterogeneous Data Sources," in *Proceedings of the 27th VLDB Conference, Edinburgh, Scotland*, 2001, pp. 241–250.

[14] L. Chen, "Semantic caching for xml queries," Ph.D. dissertation, Worcester Polytechnic Institute, 2004.

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

[16] M. F. Fernandez, D. Suciu, "Optimizing Regular Path Expressions Using Graph Schemas," in *ICDE*, 1998, pp. 14–23.

[17] J. Shanmugasundaram, *et al.*, "Architecting a network query engine for producing partial results," in *WebDB*, 2000, pp. 17–22.

[18] D. Barbosa, A. Mendelzon, and J. Keenleyside et al., "ToXgene: a Template-Based Data Generator for XML," in *Proceedings of WebDB*, 2002, pp. 49–54.

[19] M. E. Crovella, *et al.*, "Heavy-tailed probability distributions in the world wide web," in *In A Practical Guide To Heavy Tails, chapter 1*. Chapman Hall, 1998, pp. 3–26.

[20] M. Wei, *et al.*, "Utility-driven load shedding for xml stream processing," in *WWW*, 2008, pp. 855–864.

[21] N. Tatbul, *et al.*, "Load shedding in a data stream manager," in *VLDB*, 2003, pp. 309–320.

[22] B. Babcock, *et al.*, "Load shedding techniques for data stream systems." in *MPDS*, 2003.

[23] S. Al-Khalifa, *et al.*, "Structural joins: A primitive for efficient xml query pattern matching," in *IEEE International Conference on Data Engineering (ICDE)*, Feb 2002, p. 141.

[24] Y. Wu, J. M. Patel and H. V. Jagadish, "Structural Join Order Selection for XML Query Optimization," in *ICDE*, 2003, pp. 443–454.

[25] M. Wei, *et al.*, "Achieving high output utility under limited resources through structure-based spilling in xml streams," Worcester Polytechnic Institute, Tech. Rep., 2009.

APPENDIX

A. GRAMMAR OF SUPPORTED QUERIES

The grammar of the supported XQuery expressions is shown in Figure 14. A large range of common XQueries can be rewritten into this subset [13]. A query with "let" clauses can be rewritten into an XQuery without "let" clauses (by Rule NR1 in [13]). A query with FWR expressions nested within a "for" clause can also be rewritten into our supported subset format (by Rule NR_4 in [13]). The filter expression in an XPath can be moved into the "where" clause.

```
CoreExpr ::= ForClause WhereClause? ReturnClause
             | PathExpr
PathExpr ::= PathExpr "/"|"//" TagName|"*"
             | varName
             | streamName
ForClause ::= "for" "$"varName "in" PathExpr
             ("," "$"varName "in" PathExpr)*
WhereClause :: = "where" BooleanExpr
BooleanExpr ::= PathExpr CompareExpr Constant
             | BooleanExpr and BooleanExpr
             | PathExpr
CompareExpr ::= "="|"!="|"<"|"<="|">"|">="
ReturnClause = "return" CoreExpr
             |<tagName>CoreExpr ("," CoreExpr)* </tagName>
```

Figure 14: Grammar of Supported XQuery Subset

B. GENERAL FRAMEWORK FOR STRUCTURAL SPILLING

The framework we propose to address the structural spilling problem in this work is general, meaning it could be applied to any XML stream management system. Recall that to solve the structural spilling problem, we have to examine the possible spilling candidates, derive the spilling effects, measure the quality as well as cost of the reduced queries, and run the optimization algorithm to choose the optimal reduced query. As discussed in Section 4, the spill candidates are generated based on the query pattern tree, which is directly derived from the query. For each spilling candidate, determining the spilling effects in the query is resolved by deciding the data dependency relationship between the spilled path and paths in the query. Hence determining spilling effects is related to the query semantics. It is not related to the specifics of the implementation of query processing. The quality model in Appendix D measures the output quality based on the query result. Again this is solely based on the query semantics and thus general. Note that our optimization algorithms to search the optimal reduced query are cost-based approaches. Obviously, the execution cost measurement for each spilling candidate in other stream engines may be different from that of our system because of the specifics of query processing. For this, we can plug in the cost model of other stream engines. In this case, the optimality of our search algorithms can still be guaranteed.

C. ALGORITHM GENERATING MINIMUM SET OF SPILLING CANDIDATES

The algorithm that generates the minimum set of all non-redundant spill candidates is described below:

Algorithm 1 minCandidates

```
Input: Query Tree $T$
Output: candidate set $S$
void minCandidates(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
```

D. METRICS FOR QUALITY AND COST

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

D.1 Output Quality Model

Previous studies on approximate query answering tend to focus on the relational model, where the output quality is usually measured by the throughput or the cardinality [21, 22]. However, in our work, since each output result may be partial, measuring the throughput or cardinality of the output is no longer so meaningful. Here we propose a "fine-grained" 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 incorporate the cardinality of each substructure into the output quality.
2. **Shape**. Returned substructures may not be of the full shape when the corresponding paths in the 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 the query can be calculated as $S_q = \frac{Size\ of\ element\ after\ spilling}{Size\ of\ element\ without\ spilling}$ (Here we assume the size of an element is fixed).

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 a 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:

$$\nu = \sum_{n} \sum_{i=0}^{j} \sum_{q \in B(i)} N_q * S_q \quad (4)$$

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.

EXAMPLE D.1. *We calculate the unit quality of Q1 for data in Figure 3(a) (plan is shown in Figure 1). The quality of each sub-*

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 15: Quality for Q1

*structure is shown in Figure 15. For each topmost element a, a result structure <pairQ1> is returned. In this example, only one result structure is produced. Hence n=1. The result structure is composed of three substructures, $\$a//b$, $\$a/d$ and $\$a/b/c$. This indicates j=3. 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 since their c children are missing. So the output quality for $\$a//b$ is 1+2*0.0.5= 2. The unit quality for Q1 is 1+2=3.*

D.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 the 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.

Locating Cost. The locating cost indicates the cost spent on retrieving tokens. Automata are widely used for pattern retrieval over XML streams [2, 4]. The relevant tokens are "recognized" by the automata and then assembled into elements. The formed elements are passed up to the algebra plan to perform structural join and filtering. Let us use an example to illustrate the locating savings. The automaton (the automaton is augmented with a stack to keep track of the context of the tokens) for Q1 is shown in Figure 16 . When the start tag <b> matching path $/a//b$ is encountered, the automaton transitions to state $s4$. Since $s4$ is a destination state, it will invoke a flag to henceforce buffer tokens until the end token of b arrives. Similarly start tag <c> will lead the automaton to transit to state $s6$. When spilling $/a//b$, we still need to transition to state $s4$, so that we can recognize the tokens to be flushed to disk. However, the automaton does not need to transition to state $s5$ nor $s6$ since the data corresponding to $/a/b/c$ will be "automatically" flushed to disk due to spilling. In this case, the transition costs for $s5$ and $s6$ are saved. Such locating cost savings arise due to the subtree of $/a/b/c$ being contained by subtree of $/a/b$. While the detailed locating cost model is discussed in [20], we estimate the locating cost savings using the formula below [20]:

$$\sum\nolimits_{q \in A^{p_i}} n_{active}(q) C_{transit} \qquad (5)$$

Here P_i indicates the query paths whose subtrees are contained by subtrees of spilled paths. A^{p_i} denotes the set of states corresponding to P_i and its dependent states in the automaton. $n_{active}(q)$ denotes state invoking times and $C_{transit}$ denotes the transition cost. The notations are in Table 1.

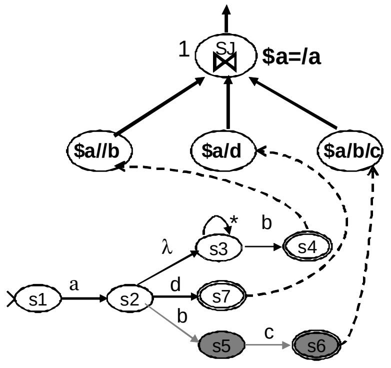


Figure 16: Locating Cost Savings When spilling $/a//b$

Notation	**Explanation**
A^{P_i}	Set of states of pattern P_i and its dependent states.
$n_{active}(q)$	The number of times that stack top contains a state q when a start tag arrives
$C_{transit}$	Cost of transition to states in automaton
N_P	Number of elements matching P for a topmost element
$S_{\bowtie}$	Join Selectivity
M_P	Size of P (number of tokens contained in each element)
C_j	Cost of comparing two elements
$C_{I/O}$	Cost of disk I/O
C_s	Cost of stack operation

Table 1: Notations Used in Cost Model

Join Cost. Since we assume stream data arrives in order, the elements for both join inputs are sorted. We can apply an efficient structural join algorithm, such as Stack-Tree-Anc [23], since both inputs are sorted. Using the cost model for this algorithm [24], we estimate the cost of structural join using the formula as below :

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

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

Spill Cost. Although join computations are saved due to spilling, we now have to consider the additional costs associated with spilling. As will be discussed in Appendix E, 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 spill cost can then be calculated as follows:

$$\sum_{p \in SP} N_p M_p C_{I/O} \qquad (7)$$

Runtime Statistics Collection. We collect the statistics needed for the costing using the estimation parameters described above. We piggyback statistics gathering as part of query execution. For instance, we attach counters to automaton states to calculate N_P and $n_{active}(q)$. And we collect M_P and $S_{\bowtie}$ in algebra operators. We then use these statistics to estimate the cost of reduced queries using the formulas given above. Note that some cost parameters in Table 1 such as $C_{transit}$, A^{P_i}, C_j and $C_{I/O}$ are constants. We do not need to measure them during the query execution.

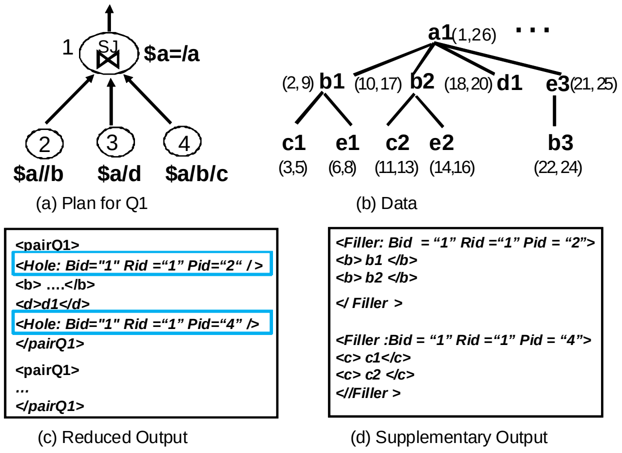
E. GENERATE SUPPLEMENTARY RESULTS

In this section, we first describe the complementary output model we propose to utilize to match the supplementary "delta" structure with partial reduced outputs produced earlier. Then we examine what extra data must be flushed to guarantee the generation of supplementary results.

E.1 Complementary Output Model

In the clean up stage, supplementary results are generated to "complement" the reduced output produced earlier. So that together these two output "pieces" can be united logically to represent the full content. Since partial result structures may be generated for each output tuple, this requires us to design an output model that can efficiently match the supplementary "delta" structure with the reduced output produced earlier. Here we propose *complementary output model*, which extends from the hole-filler model [12]. The 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 Q1 when spilling $/a/b$ are shown in Figures 17(c) and (d) respectively.

To distinguish and match efficiently between holes and fillers, we define three types of IDs, namely, 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 17(d) indicates the missing $b1$ and $b2$ for path $\$a//b$ (whose PID is 2) in the <pairQ1> element for the first BPU (a element). The second filler indicates the missing $c1$ and $c2$ for path $\$a/b/c$ (whose PID is 4) for the first BPU.



```
<pairQ1>
<Hole: Bid="1" Rid ="1" Pid="2" />
<b> ….</b>
<d>d1</d>
<Hole: Bid="1" Rid ="1" Pid="4" />
</pairQ1>
<pairQ1>
…
</pairQ1>
```

```
<Filler: Bid  = "1" Rid ="1" Pid = "2">
<b> b1 </b>
<b> b2 </b>
</ Filler  >

<Filler :Bid = "1" Rid ="1" Pid = "4">
<c> c1</c>
<c> c2 </c>
<//Filler >
```

Figure 17: Example for Output Model

E.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 determine what extra data must be flushed to disk to guarantee the generation of supplementary 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.

ID	For	Return	ID	For	Return
1	SAM	UA	7	UA	UA
2	SAM	SRAM	8	UA	SRAM
3	SAM	SAM	9	UA	SAM
4	SRAM	UA			
5	SRAM	SRAM			
6	SRAM	SAM			

Table 2: Possible Combinations Between For Binding and Its Branches

Since structural join is the core component 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 three types, namely, SRAM, SAM, or UA. There are totally 3*3 =9 combinations between the binding variable and branches. The possible combinations are listed in Table 2. Note that if "where" path is SRAM, the output is blocked. Hence we ignore this case.

Note when the binding variable is SAM, query execution is not affected. Hence cases 1, 2 and 3 can be regarded to be the same as cases 7, 8 and 9 respectively. Clearly, it is not necessary to consider case 7 since complete results are produced in this case. Finally we only need to consider cases 4-6, 8 and 9. We now list one typical case below to show how to determine what extra data to flush to disk and how to compute supplementary results. Similarly, we can generate supplementary results for other cases. The details about those cases can be found in [25].

Binding Variable is UA and Branch is SRAM. In this case, multiple branches may fall into SRAM at the same time. However, the output of the structural join of V with branch $B(i)$ is independent from the output of the structural join between V and other branches. The case that one branch operator falls into SRAM is considered first and can be easily extended to the case that multiple branches are SRAM. Assume that the binding variable V is UA and one branch $B(i)$ is SRAM. We use superscript m and d to distinguish between data kept in memory and data on disk. We represent the structural join results between the binding variable V and $B(i)$ using the following equation:

$$\begin{aligned} V \bowtie_S B(i) &= V \bowtie_S (B^m(i) \cup B^d(i)) \\ &= (V \bowtie_S B^m(i)) \cup (V \bowtie_S B^d(i)) \end{aligned} \tag{8}$$

Obviously, the results of $V \bowtie_S B^m(i)$ have already been produced by the reduced query execution. We only need to calculate the supplementary results $V \bowtie_S B^d(i)$. Hence we have to reconstruct the structural join between V and $B^d(i)$ and the extra data to be spilled is the data corresponding to the binding variable V. We use a subscript to indicate the time the data was spilled. Assume that structures V and B have been pushed k times to disk, meaning the spilled data is V_1, V_2, ... V_k and B_1^d, B_2^d, ... B_k^d respectively. As we mentioned in Section 3, the query results generated based on a basic processing unit are independent from others. We assume we spill data in batch of one or more basic processing units. We thus conclude that V_x does not need to join with B_y^d if x is not equal to y since they do not belong to the same basic processing unit. Therefore the missing structural join results between V and $B(i)$ at time k can be calculated as $V_k \bowtie_S B_k^d(i)$.

For instance, for the plan of Q1 in Figure 1, when path $/a/b$ is spilled, path $\$a//b$ is SRAM. The structural join between $\$a$ and

$\$a//b$ can be calculated using Equation 8.

F. ACKNOWLEDGMENTS

This work has been partially supported by the National Science Foundation under Grant No. NSF IIS-0414567.