WPI-CS-TR-07-06 June 2007

Load Shedding 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

Load Shedding 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 the high volume and unpredictability arrival of data streams, stream processing systems may not always be able to keep up with the input — resulting in buffer overflow and uncontrolled loss of data. Load shedding, the prevalent strategy for solving this overflow problem, has todate been considered for relational stream engines. On the other hand face additional challenges and opportunities for "structural shedding", due to the complex nested XML input and result structures. We now tackle this open XML shedding problem by a three-pronged solution. First, we develop a preference model for XQuery to enable users to specify the relative importance of preserving different subpattern in the complex XML result structure. This transforms shedding into the problem of rewriting the user query into possibly several shedding queries that return approximate query answers yet with the highest possible utility as measured by the given user preference model. Two, we develop a cost model to compare both the performance and the utility of alternate shedding queries. Third,we propose two solutions: OptShed, and FastShed. OptShed guarantees to find an optimal solution however at the cost of an exponential complexity. FashShed as confirmed by our experiments, efficiently achieves a close-to-optimal result in a wide range of cases. Lastly we describe the in-automaton shedding mechanism for Raindrop system. The experimental results show that our proposed preference-driven shedding solutions always consistently achieve higher utility results compared to the existing "relational" shedding techniques.

1. Introduction

XML has been widely accepted as the standard data representation for information exchange on the web. XML stream systems in particular have attracted interest recently [6, 10, 14, 20, 16, 22] because of the wide range of potential applications such as auction, traffic monitoring and online stores. 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 a tree structured XML element nodes. This means the engine has to extract the tokens to form the XML elements. 2) We need to do disection, restructuring, and assembly of complex nested XML elements specified by XML query expressions, such as XQuery.

For most monitoring applications, immediate online results often are required, yet system resources tend to be limited given voluminous high arrival rate data streams: 1) Sufficient memory resources may not be available to hold all incoming data or 2)CPU processing Note that for relational stream systems, tuples are the smallest granularity for shedding. However, in XML stream systems, the query result is composed of possibly complex nested structures. That means each output may be composed of a variety of elements, each of them may possibly be extracted from different positions of XML tree structure vary in their importance or processing cost. This

provides new opportunity for selectively sheding XML sub elements to achieve high processing speed. In this work, we focus on how to trade off accuracy of XML query result for performance.

In recent years, several load shedding techniques for stream system have been proposed [25, 3, 13, 9]. The current state-of-the-art in load shedding can be categorized into two main approaches. One is random load shedding [25], where tuples are discarded randomly whenever the rate of processing data cannot match the input rate and thus the output rate is significantly affected. A certain selection rate σ maybe customized and adapted according to the workload [13]. The other approach is semantic load shedding. It assigns priorities to tuples based on their utility to the application and then shed those with low priority first. Essentially semantic shedding is to shed tuples that do not contribute to generate output.

An XQuery may return query results with complex tree structures. In this tree structure, subelements may differ in their perceived importance (utility). Further, these subelements may consume rather different buffer space and require different CPU resources for their extraction, buffering, filtering and assembly. Consider an online-store, customers may have periods of heavy usage, say at some promotion time or on holidays. The online store would receive huge numbers of orders from customers. The schema for transaction element is given in Figure 1. Given a fixed buffer of size B. Assume the data arrival rate is λ. The system query processing speed is η, as is determined by the available computational resources and the query workload. When the processing capacity is not sufficient to keep up with data arrival rate, the data in the buffer will accumulate resulting in an overflow. In this case, we have to either drop some data or improve the processing speed. However, dropping complete transaction elements means that we may effectively lose some important information. In this scenario, dropping some unimportant but resource-intensive sub elements from a root element may be more meaningful to output receivers compared to the complete-tuple-granularity shedding strategy. We call this type of "element" granularity drop *structural shedding* since it changes the structure of query results. Let us consider a online store query issued below. The corresponding query pattern tree is shown in Figure 2.

```
Q1:
FOR $a in stream("transactions")/list/transaction
 WHERE $a/order/price > 100
      RETURN   $a//name,   $a//tel,   $a//email,
          $a//addr, $a/order/items, $a/survey
```

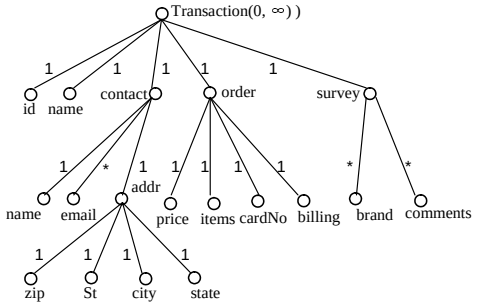

Figure 1. The schema definition for Q1

This query is to return the customer's contact information and item list when their transactions are spend more than 100 dollars. The contact information including customer's telephone, email and the items they bought. To process as many transaction tuples as possible, output receiver may prefer to selectively obtain partial yet important content in the query result while dropping less important subelements in each transaction tuple. In this case we may choose to drop "addr" information for two reasons: 1) "addr" element is much more complex than "email", as

can be seen in the schema. This means we have to process more tokens for each single “addr” element; 2) ”addr” element may be ”optional” to output consumer because “email” may be the more likely means of contacting customers. By dropping the ”addr” element, several savings arise. First, we do not need to extract ”addr” element from the input tokens. In this case, we save the processing cost of locating tokens from “<addr>” to “</addr>”. Second, we no longer need to buffer “addr” element during query processing. Thus the buffering costs for ”addr” element is saved. Note here the query is changed to a new one due to removing the “addr” element. Let us call the new reduced query *shed query*.

There are many options to drop subelements based on the query. However, different shed queries vary on their importance and their processing costs. Hence choosing an appropriate shed query is very important. This raises many challenges. First, given a query, what are different ways to change the query via shedding while keeping the query valid. Second, what model to we employ to specify the importance of each subelement. Third, which of the potential shed query to choose to obtain maximum utility, and lastly how to implement structural shedding in this XML context. Our solution tackles these challenges taking a three-pronged strategy. One, we develop a preference model for XQuery to enable output consumers or user who issue the query to specify the relative *utility*(a.k.a preference) of preserving different subpattern in the query. By comparing the utility of different shed queries, we can judge which one yields highest utility, i.e. satisfies consumer’s preference best. Two, we develop a cost model to estimate the processing cost for the candidate shed queries. The main goal of our shedding technique is to maximize output utility giving the input rate and limited computational resources. We propose two solutions: Utility-optimal, and Ratio-based Greedy (RG). Utility-optimal guarantees to find an optimal solution however at the cost of an exponential complexity. RG as confirmed by our experiments, achieves a close-to-optimal result in a wide range of cases. Lastly we discuss the implementation of an in-automaton shedding mechanism in the Raindrop systems.

Our contributions are summarized as below:

1. We introduce the concept of structural shedding for XML stream systems. To our best knowledge, we are the first to address shedding in the XML stream context and to exploit the utility for XML elements into shedding decision.

2. To solve the shedding problem, we introduce two classes of algorithms, Utility-Optimal, and Ratio-based Greedy (RBG).

3. We propose a simple yet elegant mechanism for performing shedding in our query engine at run-time, namely, suspending the appropriate states in the automaton-based execution engine.

4. We provide a thorough experimental evaluation that demonstrates that our approach maximizes the utility while keeping the CPU costs under the system capacity.

2. Background

2.1. Query Pattern Tree

We support a subset of XQuery. Basically, we allow “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. Here we assume the queries have already gone through the normalization steps in [7].

The query pattern tree for query Q1 is given in Figure 2. In Figure 2, each navigation step in an XPath is mapped to a tree node. We distinguish between three types of nodes: context nodes, return nodes and select nodes

as indicated by annotations in Figure 2. First, *context node* is the node that corresponds to a context variable in the "for " clause, e.g., *$a* in Figure 2. Context nodes must evaluate to a non-empty set of binding for the FWR expression to return any result. This implies that the query would return nothing if we drop the context node. In this case, dropping a context node would have the same result as dropping the root element. Second, the nodes that correspond to the pattern in the "return" clause, e.g., *item* or *category* are called *return node*. Note that return nodes are optional patterns meaning even if $\$a$//*tel* evaluates to empty, other elements will still be constructed. The third type of the nodes correspond to the pattern in the "where" clause. We call such nodes *selection nodes*. For instance, if an XPath is an operand in a comparison predicate, then the destination node of the XPath, e.g., *price* of $a/order/*price* in Figure 2, is a selection node. We add annotations on the nodes to indicate their types. A context node is annotated with "c". A return node is annotate with "r " and a selection node is annotated with "s".

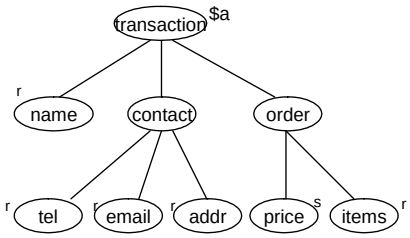

Figure 2. The corresponding pattern tree for Q1

Here we define destination nodes to be either "return" or "where" clause *patterns*. Particularly we call return nodes "r" patterns and selection nodes "s" patterns. In the query pattern tree shown in Figure 2, the "name", "email", "tel", "items" and "survey" are "r" patterns. The "price" node is an "s" pattern.

2.2. Generating Shed Queries

We now investigate the different shed queries that can be generated for one giving query via shedding. Clearly we should avoid generating sub queries that would result in an increased output rate. Thus randomly choosing an element to drop may not be meaningful. For instance, in Figure 2, dropping element $\$a$ is not a good idea because $\$a$ is the outer loop binding variable. Thus this drop would cause the resulting shed query to return nothing. That is, dropping the element corresponding to $\$a$ is equal to dropping the root element.

The shed query generation process being based for an original query Q0 has to follow three construction rules:

1. Only "r" and "s" nodes are allowed to be removed from the original tree T_0.
2. Any shed subtree T_i always has the same root as T_0.
3. The leaf nodes of a subtree have to be either "r" node or "s" nodes. For instance, the subtree depicted in Figure 3 is not allowed. This tree does not need to keep the "contact" element because all children of the "contact" element are removed and it is neither an "r" nor an "s" pattern.

Assume B denotes the number of all "r" and "s" patterns for a given query tree. When the query tree is very bushy of width B, we can generate the maximum number of shed queries 2^B. When the query tree is deep and

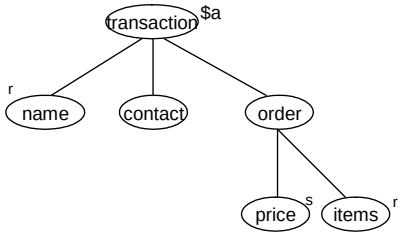

Figure 3. Not Acceptable Query Tree

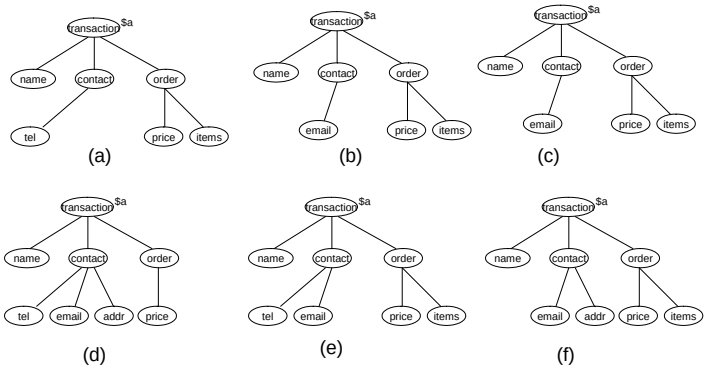

Figure 4. Some shed query trees

linear, we would generate at most B shed queries. Thus the number of shed queries for a query can vary between B and 2^B. Figure 4 shows some shed queries for query Q2.

3. XML Stream Systems: Processing and Cost Model

In this section, we describe the widely-adopted automata processing model for XML streams, which we assume as underlying model for our systems. We then design a cost model for this XML stream processing model. As is known, automata are widely used for pattern retrieval over XML token streams. [10, 20]. After pattern retrieval, the relevant tokens are assembled into tuples to be further returned or filtered as final output elements. The formed tuples then passed up to perform structural join and filtering. A plan for query Q1 is shown in Figure 5. Observe that the context node $\$a$ in the "for" clause is mapped to a structural join in the plan. Thus we have the following query processing tasks in XML stream systems:

1. Locating tokens. We use an automaton to retrieve the patterns.
2. Extracting tokens. After retrieving the tokens, we extract the tokens and compose them into XML tuples.
3. Manipulating buffered data. Algebra operator provide XQuery translation functionalities, including structural join and selection.

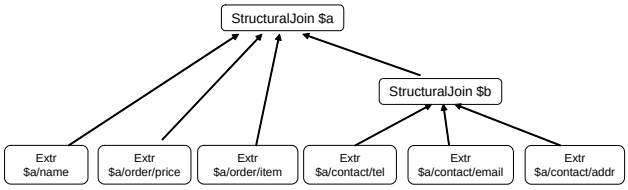

Figure 5. An Example Plan

For instance, in query Q1, we perform structural join on $a to collect all pieces to form a transaction tuple. In addition we perform selection on $a/order/price to judge whether the "price" is greater than 100.

3.1. Automaton-based Implementation

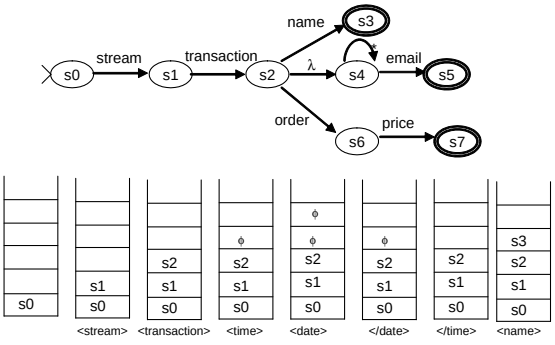

Figure 6. Snapshots of Automaton Stack

We briefly describe how the automaton functions. A stack is used to store the history of state transitions. The bottom of Figure 6 shows the snapshots of the stack after each token is processed. Initially, the stack contains only the start state $s0$ (see the first stack). The automaton behaves as follows:

- **When an incoming token is a start tag**:

  When we see a start tag, we need to check whether this start tag will lead us to any transitions in the automaton. There are two possibilities, either such next state exists or not. In the first case, we transition to a new state. Within this state, tasks to be undertaken may include setting a flag to henceforth buffer tokens, such as to record the start of a pattern, trigger a structural join, etc. We call this cost $C_{transit}$. Note that the start tokens of all elements in the query tree will cause such a transition. The other case is that there are no states to transition to. In this case, an empty state is simply pushed onto the stack top without any other actions. For instance, when $<time>$ is encountered, the stack pushes an empty state onto the top. Note that all start tags of patterns that do not appear in the query tree will lead to such a transition. The cost associated with this case is C_{null}.

Notation	Explanation
N^{P_i}	Number of elements P_i for topmost element.
N^{start}, N^{end}	Total number of start or end tags for a topmost element.
S^{P_i}	Number of tokens contained for a P_i element.
Q^A	Set of states in automaton A.
Q^{P_i}	The set of states which include the state corresponding to pattern P_i and all its following states.
$n_{active}(q)$	the number of times that stack top contains a state q when a start tag arrived
$C_{transit}$	cost of processing a start tag of an element in the query tree
C_{null}	cost of processing a start tag of an element not in the query tree
$C_{backtrack}$	cost of popping off states at the stack top
$C_{buf}(q)$	cost of buffering a token
$C_{join}(e)$	cost of performing a structural join on a single element e
$C_{sel}(pred_i)$	cost of evaluating predicate $pred_i$
$\sigma(e)$	selectivity for predicate evaluation on all elements e for a bottom input element

Table 1. Notations Used in Automaton Costing

- **When an incoming token is a PCDATA token**:
  The automaton makes no change to the stack.

- **When an incoming token is an end tag**:
  The automaton pops off the states at the top of the stack (see the sixth stack when $</emph>$ is processed). We call such popping off cost as $C_{backtrack}$. Note that the popping cost for all end tags is the same, regardless of if thestack top is empty or not.

3.2. CPU Cost Model for a Query

Traditional database models defined the cost of a plan as the processing time of the entire input data. However for XML stream processing, this is not possible, as the stream could potentially be infinite. One solution here is to define the cost on a finite input chunk while the entire stream can be seen as a sequence of the chunks. It is natural to consider a complete topmost element (except the start and the end of the whole XML stream) since it is the basic unit based on which we generate query results. We call the processing time of handling such a top most element the *unit processing cost*. We measure the cost of a query based on its unit processing cost. For instance, the cost of query Q1 thus is the unit processing cost of handling one $transaction$ element.

Based on the above analysis of the basic functioning of an automaton-based implementation, we divide the processing cost (UPC) for XQuery into three parts: Unit Location Cost (ULC) that measures the processing time spent on automaton retrieval, Unit Buffering Cost (UBC) spent on pattern buffering and Unit Manipulation Cost (UMC) spent on algebra operations including selection and structural join. The relevant notations are given in Table 1. The Unit CPU Cost for a query Q_i can be written as below:

$$UPC = ULC + UBC + UMC \quad (1)$$

3.2.1 Unit Location Cost (ULC)

We split the total ULC into two parts, one part considers the cost of locating the start and end tags for elements in the query tree, and the other part considers the cost for locating the start and end tags for other elements. The

first part can be measured by considering the invocation times for each state and the transition cost for a token as below:

$$\sum_{q \in B^(A)} n_{active}(q)(C_{transit} + C_{backtrack}) \tag{2}$$

$\sum_{q \in B^(A)} n_{active}(q)$ denotes the number of times new states are transitioned to for start tags of all elements in the query tree. The number of other start tags, namely for elements which are not in the query tree, can be written as $n_{start} - \sum_{q \in B^(A)} n_{active}(q)$. Thus the second part of the transition cost is as below:

$$(n_{start} - \sum_{q \in B^(A)} n_{active}(q))(C_{null} + C_{backtrack}). \tag{3}$$

In total, the ULC for a given automaton A is:

$$\begin{aligned} & transition\ cost\ for\ all\ tokens\ in\ a\ bottom\ input\ element \\ = \ & transition\ cost\ for\ tags\ in\ query \\ & + \ transition\ cost\ for\ other\ tags \\ = \ & \textstyle\sum_{q \in B^(A)} n_{active}(q)(C_{transit} + C_{backtrack}) \\ & + \ (n_{start} - \textstyle\sum_{q \in B^(A)} n_{active}(q))(C_{null} + C_{backtrack})) \end{aligned} \tag{4}$$

3.2.2 Measuring Unit Location Cost (ULC) Savings for SubQueries

We now look at how to estimate the location cost we can save by switching from the initial query to a shed query. Assume the shed query is the query with pattern p_i chosen to be dropped. This means that the pattern p_i and all its descendant patterns will be dropped. Then in the automaton for the shed query, as the state corresponding to p_i and all its subsequent states will be cut from the initial automaton A of Q. Let us call the set of states corresponding to p_i and its dependant states B^{P_i}. The location cost for pattern p_i in initial automaton can be represented as:

$$\sum_{q \in B^{p_i}} n_{active}(q)(C_{transit} + C_{backtrack}) \tag{5}$$

However, in the automaton for the shed query, since these patterns are now treated as elements that are not in the query. Their location cost is now changed to:

$$\sum_{q \in Q^{p_i}} n_{active}(q)(C_{null} + C_{backtrack}) \tag{6}$$

Thus the savings in location costs gained by switching from the initial query to the shed query can written as:

$$\begin{aligned} & \textstyle\sum_{q \in Q^{p_i}} n_{active}(q)(C_{transit} + C_{backtrack}) - \\ & \textstyle\sum_{q \in Q^{p_i}} n_{active}(q)(C_{null} + C_{backtrack}) \\ = \quad & \textstyle\sum_{q \in Q^{p_i}} n_{active}(q)(C_{transit} - C_{null}) \end{aligned} \tag{7}$$

For instance, suppose the pattern “name” is chosen to be dropped from the shed query. Then in the automaton for the shed query, state $s3$ will be cut from the automaton for the initial query. Here the the number of times that state $s3$ was invoked is equal to the number of start tags of “name”. The savings are:

$$\begin{aligned} & \textstyle\sum_{q \in Q^{name}} n_{active}(q)(C_{transit} - C_{null}) \\ = \quad & N^{name}((C_{transit} - C_{null}) \end{aligned} \tag{8}$$

3.2.3 Unit Buffering Cost and Saving

In our query engine, not all incoming tokens are stored. We only store the tokens that are required for further processing of the query. For a given query, all "r" and "s" patterns are to be stored, since they need to to be returned or filtered. We associate such pattern information with their corresponding states in the automaton. For example, in Figure 6, state $q4$ represents an "r" pattern. Note that it is associated with the algebra node $Extract_{\$a}\$b$. Once $s4$ is activated by the arrival of $<name>$, $Extract_{\$a}\$b$ raises a flag. As long as the flag is raised, the incoming tokens will be stored to compose the *seller* element nodes. When $s4$ is popped off the stack by the arrival of end tag, i.e., a *</name>*, $Extract_{\$a}\$b$ revokes the flag and thus terminates the extraction of the *name* element. We assume that for each individual token, the buffer cost is fixed. The buffer cost for a topmost element is defined as UBC (Unit Buffering Cost).

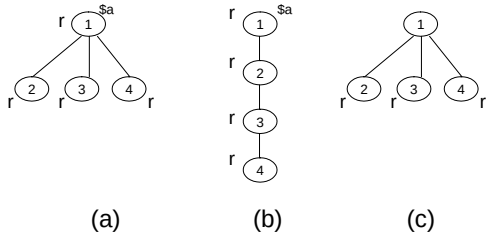

Figure 7. Buffer Sharing Examples

We do not store the same token twice in our buffer, instead they are shared. Three query examples are shown in Figure 7. In Figure 7(a) and 7(b), the parent pattern and its children overlap. Since both the parent and the children are to be returned, we only need to store the parent pattern $p1$ and set a reference for its children $p1$, $p2$ and $p3$ pointing to $p1$. In this case, the buffer cost is equal to the buffer cost of the parent pattern $p1$. However, in Figure 7(c), since the parent is not a "r" pattern, only its children are to be returned. The buffer cost is equal to the buffer cost of all the children.

Hence, for a given query, we need to find all the non-overlapping topmost patterns required to be buffered, called henceforth the *storing pattern set*. To obtain this storing pattern set, we traverse the tree from the root node in a breadth-first manner. If the root node is an "r" pattern, then add this root node to the storing pattern set and stop here. Otherwise, for each its children, check from left to right. if the node is a "r" pattern or an "s" pattern, add this node into storing pattern set and label all its descendants as visited. If it is not, we move on and check its children, repeating the above steps.

Assume the storing pattern set for our query Q is denoted as R. UBC can be written as

$$UBC(Q) = \sum_{p \in R} N^p S^p C_{buf} \tag{9}$$

For instance, the storing pattern set for query tree in Figure 7(a) is $\{p1\}$. However the storing pattern set for Figure 7(c)is $\{p2, p3, p4\}$. For the query tree shown in Figure 7(a), if pattern $p2$ is chosen to be dropped, no buffering cost is saved since the buffering cost for the query is equal to the buffering cost of pattern $p1$. However, for the query tree shown in Figure 7(c), if pattern $p2$ is chosen to be dropped, the buffering cost for the new query will be reduced by $N^{p2} S^{p2} C_{buf}$.

3.2.4 Unit Manipulating Cost (UMC) and Measuring Manipulating Cost Saving

The Unit Manipulating Cost is defined as the cost spent on selection and structural join operations(the cost of Extract operator is measured by the buffer cost). Note that in the plan shown in Figure 5, the algebra operators consist of selection and structural join operators. For each operator in the plan, we must estimate the cost of performing such an operator. In addition, we must estimate the size of the result for each operator, since this result is the input for the downstream operators. Note that the leaves of the plan can only be Extract operators since tokens for each pattern have to be extracted first. Here we denote the cardinality of the output from each Extract operator on pattern p_i N^{p_i}. We now look at how to estimate the unit cost of the whole plan. First we will look at how to estimate the cost and result size of the selection operator. Then we will examine how to estimate the cost and result size of a structural join operator.

For a selection pattern p_s in the query, several predicates may be defined on it. We define two concepts for selection operators, selectivity and non-empty probability. The selectivity for predicate $pred_i$ on pattern p_s is defined as

$$\sigma^{P_s}(pred_i) \ = \ \frac{Number\ of\ elements\ satisfying\ \ pred_i}{Number\ of\ input\ elements} \tag{10}$$

For each selection operator, we need to estimate whether each element of pattern p_s satisfies the predicate $pred_i$. We call this result non-empty probability. As long as at least one element of pattern p_i is evaluated to be true, the result of this selection operator is true and we can stop checking other elements. In other words, the return result for selection operator is equal to the disjunction of evaluation result for each element. Thus the non-empty probability for selection on a predicate $pred_i$ is defined as:

$$\begin{aligned} & p_{\neq\phi}(pred_i) \\ = \ & \bigcup_{j=1}^{N_{in}} P(No\ jth\ element\ satisfies\ pred_i) \end{aligned} \tag{11}$$

If the pattern p_s needs to be returned, we also need to count how many elements of pattern p_s satisfy the predicate $pred_i$ among the input elements, in other words, the result size. Assume there are $i-1$ predicates located upstream of selection operator on $pred_i$. Note that the input cardinality for the bottommost selection operator is N^{p_i}. Its result size of selection $pred_i$ is equal to:

$$N_{out}(pred_i) \ = \ N^{p_i} \prod_{k=1}^{i} \sigma^{P_s}(pred_k) \tag{12}$$

The cost for a selection operator of pattern p_s on $pred_i$, $C_{sel}^{p_s}(pred_i)$, is decided by the number of input elements for the operator and unit predicate evaluation cost on one predicate. The selection cost for predicate $pred_i$ on pattern p_s can be written as:

$$C_{sel}^{p_s}(pred_i) \ = N_{in}^{p_s} C_{sel}(pred_i) \tag{13}$$

The structural join cost is decided by the types of input operator and the number of the elements in each input. If the structural join has a selection operator descendant, it needs to consider the selectivity of the the selection operator before since the structural join is performed only when the predicate is evaluated to be true. Thus the cost of the structural join on the element e $C_{sj}(e)$ can be defined as below:

$$C_{sj}(e) \ = (\textstyle\prod_{B \in I^e} N^B) * C_{join} \tag{14}$$

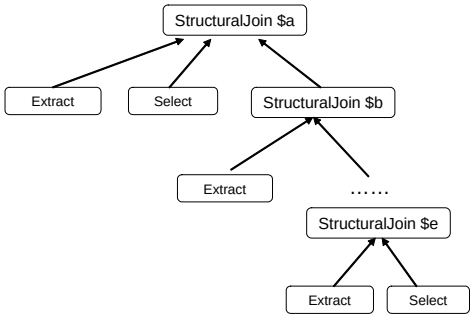

Figure 8. Structural Join Cost Example

Note that I^e denotes the input operators of structural join operator on element a and N^B denotes the number of elements in input operator B. C_{join} is the processing time for joining one element from one input.

The result size of a structural join operator on element e is decided by how many elements are bound to e in a top most element and whether each element e successfully generates non-empty query result when it has non-returned selection operator. Assume there are N^e elements bound to e, the result size of a structural join operator on e is thus written as:

$$\sum_{k=1}^{N^e} \prod p_{\neq\phi}(B), here\ B\ \in I^e\ and\ B\ is\ selection \tag{15}$$

For the shed queries, we can estimate the unit manipulation cost savings by checking which input pattern is dropped in the shed query. Suppose the pattern P_k is dropped from the shed query, the Extract operator and the selection operators $pred_1$... $pred_i$ are then removed from the plan. Assume there is one structural join on top of it, the UMC savings in the new plan can estimated by:

$$\begin{aligned} \delta(UMC) &= C_{sel}^{p_k}(pred_1) + C_{sel}^{p_k}(pred_1) + ...C_{sel}^{p_k}(pred_i) \\ &= (\prod_{B\in I^e} N^B - \prod_{B\in\{I^e - pred_i\}} N^B) * C_{join} \end{aligned} \tag{16}$$

When there is no predicate under the structural join, the selectivity $\sigma(pred)$ is set to 1 default. For example, there is a structural join on pattern $p1$ shown in Figure 7. Assume we drop pattern $p2$ in the shed query, the structural join cost for the shed query is equal to $N^{p3} * N^{p4} * C_{join}$ instead of $N^{p2} * N^{p3} * N^{p4} * C_{join}$ in the initial query.

4. Runtime Statistics Collection

We now sketch how we collect the statistics needed for the costing using the estimation methods described in Section 3. We piggyback statistics gathering as part of query execution. We attach counters to automaton states and algebra operators to collect the statistics. During query execution, statistics collection for an automaton-based operator for instance proceeds as follows: when a state q is activated, its associated counter will be incremented to collect $n_{active}(q)$. Then for k most recent tompmost elements, we combine the statistics gathered in this period with the last statistics using a weighted function that gives higher priority to the more recently collected statistics

over older ones. We then use these statistics to estimate the cost of the candidate sub queries using the formula given in Section 3.

Note that some of the cost parameters in table 1 such as $C_{transit}$, C_{null}, C_{buf} and C_{join} are constants. We do not need measure them in the query execution. Other parameters, namely, cardinality and counts such as $n_{active}(q)$, N^p, S^p and n_{start}. Note that N^p is actually equal to $n_{active}(q)$ where q is the corresponding states for pattern p in the automaton.

5. Preference Model for Query Pattern Trees

Different elements may vary in their importance to real applications. For instance, the "name" element may be more important than the "email" element for query Q1 because "name" is the unique identifier of a $transaction$ while "email" may in fact be an optional item for $transaction$. Thus we need a metric to measure the importance of each pattern for a given query. In this work, we define a quantitative preference model to represent preference of different elements for a query. THe preferences can be specified by the user who issue the query or the output consumer. By binding different nodes with their respective preference, shed queries would vary in their overall perceived utility to the user. In this way, we can rank the queries derived from the initial query. The notion of preference model has been investigated in the literature [18]. Instead of using the preference model to represent the preference between tuples as in previous work [18, 12, 8], we are using the preference model to represent the preference between different nodes in the query tree. We support two types of preferences representations, one using prioritized preference [17] to explicitly express the relevant ranking among different elements, and the other is using a quantitative approach [12, 11] that by scoring function to represent the importance for the nodes. In this work, we allow user to choose either model to represent their preference. For prioritized preference model, we provide a default score assignment to assign scores to different nodes.

Before we describe our proposed preference model, we first analyze what nodes user needs to assign preferences. As we mentioned in Section 2.1, there are three types of nodes in the query tree. Recall that 'c" nodes are considered essential, i.e., they cannot be shed. So we only need to consider the preference of the "r"(return nodes) and "s" patterns (selection nodes).

5.1. Prioritized Preference Model(PPM)

If a user chooses to use the prioritized preference, they describe the partial-order relationship between nodes. It means that given a query tree, for all the nodes on the query tree, the user has to declare the ordering of all "r" and "c" nodes in term of their importance.

An example prioritized preference for query Q1 could be:

$name \succ order \succ price \succ items. \succ tel \succ email \succ survey$

Note that prioritized preference satisfies the structural importance relationship, that is, the parent pattern is more important that its children pattern. This is because the pattern always contains its child pattern. After ranking, a default score assignment strategy is applied based on relevant ranking. The score calculation formula is given as:

$\nu(Pattern\ Ranking\ k) = \frac{1}{2^k}$

For instance, the utility for "name" is equal to $\frac{1}{2}$ and the utility of order is equal to $\frac{1}{2^2}$. Note this score assignment method can conserve the structural importance relationship since it guarantees that the utility of pattern ranking k is greater that the sum of utility for patterns ranking after k.

5.2. Numerical Preference Model (NPM)

If users choose to use a quantitative approach, they can assign their customized importance (a.k.a utility) for different elements in the query in a numerical form. We define the scoring function to represent the importance of each node in the query tree. The scoring function for a pattern in the query tree is defined as below:

v(P_i) $\mapsto [0, 1]$

where P_i is an "r " or an "s" node in the query tree and v(P_i) is a constant value between (0,1). An example of utility for query Q1 is shown in Figure 9.

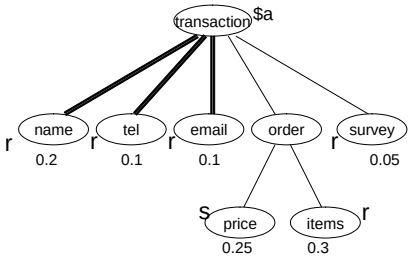

Figure 9. The Query Tree Augmented by Preference

Based on the literature [26], we can easily extend the syntax to integrate the preference into the query below:

```
Q1:
FOR $a in stream("transactions")/list/transaction
 WHERE $a/order/price > 100
      RETURN   $a//name,   $a//tel,   $a//email,
           $a//addr, $a/order/items, $a/survey
  with ν(name)= 0.2, ν(tel)= 0.1, ν(email)=0.1...
```

In some cases a user may not be able to or want to specify the scores for all nodes. Thus only some scores of "r" and "s" nodes are determined. In this case, we would consider the other nodes to be not as important as the scored nodes. Thus the scores of other nodes will be set to "not important" being represented by the infinitely small number ϵ.

When no scores for *any* nodes are specified, then we would assign default scores for all the "r" nodes and "s" nodes. In this case, the default score of all "r" nodes and "s" nodes would be set to 1.

The utility of a query pattern tree indicates the amount of utility we expect to gain by running this particular query on a topmost element. It is defined as below:

$\nu(Q_i) = \sum_{P_j \in S_i} \nu(P_j)$.

The query tree with preference for query Q1 is shown in Figure 9. In this case, the utility for Q1 is equal to:

$0.2 + 0.1 + 0.1 + 0.25 + 0.4 + 0.05$= 1.1

In addition, we have two special queries which are also considered to be shed queries: empty query Q_0 and initial query. The utility for the initial query can be calculated easily. The empty query indicated dropping the whole root element. This is equal to shed all patterns. Thus we have $\nu(Q_0) = 0$.

5.3. Total Data Utility

After discussing the utility for a query running on single topmost element, we now examine how much total utility we can gain by apply a query given some input data. For instance Q1 is applied to some topmost elements while Q2 is applied to other topmost elements. Assume totally m shed queries are chosen in shedding phase. There are x_0 topmost elements that execute empty query(dropped), x_1 topmost elements that execute Q1, x_2 topmost elements that execute Q_2, and so on. The total utility for n topmost elements can be written as below:
$\phi = \sum_{i=1}^{j} x_i * \nu(Q_i)$, where $\sum_{i=1}^{j} x_i$ = n.

5.4. User Preference Model to Prune Candidate Query Set

As we discussed in Section 2.2, when the initial query Q_1 is in a "bushy" shape, for instance, all k patterns are independent, 2^k possible shed queries where k is the number of patterns in Q_1 exist. However, some of them may contain too few patterns and thus not be meaningful to users. One idea is to use the preference model to prune these shed queries. We can set up a threshold utility value for candidate queries, e.g., half of the utility of initial query Q_1. By setting up this threshold, we can avoid generating the queries with low utility values. The other advantage of applying the utility threshold on the candidate query set is that we guarantee that our query result in the shedding have some lower-bound on the utility for each output result, similar to satisfying the accuracy on some degree.

6. CPU constraint shedding

We allow different tuples executing different queries in shedding phase, for instance, we can let first 500 tuples executing initial query Q_0, the later 300 tuples executing another shed query Q_1. First, we describe when to trigger structural shedding. Then we describe choose shed query set. o choose so to maximize the total utility. We present two solutions, OptShed and FastShed. Finally the advantages and disadvantages of these two approaches are analyzed.

6.1. Decide When to Shed

We assume a fixed memory buffer to store the input XML data. As long as all the tokens in an XML element are processed, we clean those tokens from the buffer. We set a buffer threshold for the system. From the beginning of the execution, we have load monitoring step to check the current memory buffer periodically. As long as the buffer occupancy exceeds the threshold, we would trigger shedding phase.

6.2. Problem Statement

Given the candidate shed query set $\{Q_\epsilon, Q_0, Q_1, ..Q_n\}$ where Q_ϵ is empty query and Q_0 is the initial query. Note utility of empty query Q_0 ν_0 and the UPC of Q_0 are both assume to be zero. We have the following inputs for our cpu-oriented shedding problem: 1. data arrival rate λ; 2. utilities of candidate query set $\{\nu_0, \nu_1, ..\nu_n\}$. 3. processing cost of candidate query set $\{C_0, C_1, ..C_n\}$.

Our goal is to find a coefficient vector $\{x_\epsilon, x_0, x_1, ..x_n\}$for candidate shed query set, to make the utility of the total processed XML tuples maximal while keeping the processing cost below the CPU processing capability. The formal problem can be represented below:

We have the following constraints:

1. The total number of processing XML elements(including topmost element who run empty query Q_ϵ) equal to number of input XML elements:

$\sum x_i \ = \ \lambda$

2. Total execution cost should be less than or equal to unit time.

$\sum_{i=0}^{j} x_i * C_i \ = \ x_0 * C_0 \ + \ x_1 * C_1 \ + .. + \ x_j * C_j \ <= \ 1000ms$

3. The number of tuples running query $Q_i$process cannot exceed its processing speed η_i (Except empty query whose unit processing cost can be looked as zero).

$x_i \ <= \frac{1}{C_i}$

For query Q_ϵ, its coefficient x_0 should be less than arrival rate. Because the number of dropping topmost elements cannot exceed arrival elements, meaning $x_\epsilon \ <= \lambda$

Given the above constraints, we want to maximize the total data utility:

$max\{\sum_{i=0}^{n} x_i * \nu_i\}$

Observe that the objective function is linear, i.e., it has two linear constraint functions. The variables all have be to non-negative integers. We thus conclude that this problem is an instance of the bounded knapsack problem. We have two solutions for this problem, one is dynamic programming approach, the other is greedy approach.

6.3. OptShed Approach

The OptShed solution is using a dynamic programming solution from [23]. To state our approach, we construct a matrix of sub-problems:

$$\begin{matrix} \psi_0(0) & \psi_0(1) & ... & \psi_0(1000) \\ \psi_1(0) & \psi_1(1) & ... & \psi_1(1000) \\ & ... \ ... & & \\ \psi_n(0) & \psi_n(1) & ... & \psi_n(1000) \end{matrix}$$

Here $\psi_j(\tilde{c})$ only uses queries from Q_0 to Q_j and its total cost satisfies $\sum_{i=0}^{j} x_i * C_i \leq \tilde{c}$. Clearly, $\psi_n(1000)$ is the original problem we want to resolve.

Now, we define $\phi_j(\tilde{c})$ to be the maximum utility of sub-problem $\psi_j(\tilde{c})$, the dynamic programming approach can be presented recursively as follows:

$\phi_j(0) \ = \ 0 (0 \leq j \leq n)$

$\phi_j(\tilde{c}) =$

$$\mathbf{max} \begin{cases} \phi_{j-1}(\tilde{c}) \\ \phi_{j-1}(\tilde{c} - kC_j) + k\nu_j \quad (1 \leq k \leq \lfloor \tilde{c}/C_j \rfloor) \end{cases}$$

Here the maximum utility is set to zero initially. Tabulating the results from $\phi_0(0)$ up through $\phi_n(1000)$ gives the solution. Each time the algorithm will check which sub query would contribute to the maximum utility if it is added.

Since the calculation of each $\phi(\tilde{c})$ involves examining n query pattern trees (all of which have been previously computed), and there are 1000 values of $\phi(\tilde{c})$ to calculate, the running time of the dynamic programming solution is thus O($1000n$). The time complexity for the dynamic programming can be improved to: $O(\sum_{j=1}^{n} \lfloor \log_2(1000+1 \rfloor)$ according to [23].

6.4. FastShed Approach

Since the time complexity of OptShed is big, we want to find a simple but effective way to solve this problem. FastShed, Greedy approach is proposed to solve this problem in this case. FastShed approach is described as follows:

1. For every candidate query $Q_0, Q_1, ...Q_n$, compute their utility gain ratio $\gamma(Q_i) = \frac{Utility\ of\ query\ tree\ Q_i}{Processing\ cost\ of\ query\ tree\ Q_i} = \frac{\nu(Q_i)}{C_i)}$

2. Sort the utility gain ratio in descending order.

3. Choose the query Q_{max} with the highest utility gain ratio γ_{max} as the new query. Assume the UPC of query Q_{max} is equal to C_{1st} The number of the root elements we are going to adopt this query is equal to $x_{max} = \lfloor \frac{1000000}{C_{max}} \rfloor$

4. Calculate the remaining processing cost, which is equal to

   $1000000 - x_{max} * C_{max}$

   Find the queries which have the cost lower than $1000000 - x_{max} * C_{max}$. Pick the query with highest utility gain ratio γ_{second} among these queries. The number of the root elements for this ratio is equal to $x_{second} = \lfloor \frac{1000000 - x_{max} * C_{max})}{C_{second}} \rfloor$

Since the greedy approach always use the query with the highest utility gain ratio. The time complexity is only the sorting cost. Thus the time complexity is O(nlgn), where n is the number of candidate subqueries.

7. Shedding Mechanism Implementation

In this section, we examine the implementation of different shedding approaches in Raindrop Systems [15]. For streams systems, one common implementation way is to insert drop boxes in plan [25, 2, 4]. However, in XML stream system, many systems are using automata to recognize relevant elements from arriving data. We adopt a unified framework which combines both automata and algebra plan. Instead of dropping tuples directly, we need to look at which token is coming into the system and what to do with the arriving token. We provide a state-disabling strategy which is able to drop tokens by disabling the state invocation. We then discuss how to switch shed query from one to another.

7.1. In-Automata Shedding Mechanism: Disable Transition and propagate Drop Signal

For our shedding approaches, we use an state-disabling approach to drop tokens. Assume we want to drop pattern \$a/name and \$a/contact/tel. Figure 10 shows where to insert drop box in automata. To drop pattern \$a/name, the automaton would temporarily removed the transition from state s2 to s3. When the start tag of name element comes, the state s3 is never reachable. Thus it would not invoke its downstream operator: $Extract\$a/name$. On the other hand,$Extract\$a/name$ will be labeled with a "dropped" flag. This flag guarantees that the downstream $structuraljoin\$a$ operator work correctly. Thus when $structuraljoin\$a$ check its input operator one by one, if an input operator is labeled with "dropped " signal, $structuraljoin\$a$ would skip this input.

7.2. Random Shedding in XML Streams

In XML streams, we perform random shedding dropping at automaton level. because it conforms to the "the earlier dropping, the better" rule. since the unit of incoming data in XML Streams is a token, the start token of the topmost elements is recognized by automaton. We then can set "shedding" flag to be true. As long as this flag is true, the incoming token is dropped. At the same time, we add a counter to monitor how many topmost elements we have dropped. When the end token of the topmost element arrived, the counter's value is check. If the counter's value is reached, the flag is disabled and system can switch back to "non-shedding" phase.

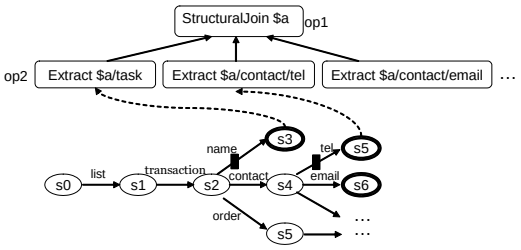

Figure 10. Disable Transition Strategy for XML Shedding

7.3. Shed Query Switching at Run-time

We support the mixture of shed queries in execution. Assume the OptShed approach provide a solution vector, say $\{60, 10, 20\}$. In this case, we will first drop 60 topmost elements, then run query Q_0 for 10 topmost element, then switch to query Q_1 for the next 20 topmost elements. What we do is using a counter to record the number of topmost elements have been run for query Q_i. When the number of topmost elements x_i has been reached. After processing the last end tag of $x_i th$ element, the system restore the removed state transition immediately and then switch to the new shed query. Since the switching time only happens after the processing of the last token of the topmost element, meaning the output has been generated, it is safe to switch to another query for next topmost element. Furthermore, we only involve with the state transition disabling and labeling "dropped" flag, we do not physically change the plan. Thus the overhead is very small.

8. Related work

In streaming systems, approximate query processing has been considered an effective method for trading off performance with accuracy[21, 25, 9, 24]. However, no approximate query processing has been touched in XML streams. Load shedding and sampling data are two most common ways to reduce system workload. Load shedding on streaming data has firstly been proposed in the Aurora system [25]. This work introduces two types of load shedding: random and semantic load shedding. Based on the analysis of the loss/gain rate, the random load shedding strategy will determine the amount to shed to guarantee the output rate. For semantic drop, they assume that different tuple value may vary in term of utility to application. In this case, maximizing the utility of output data is their goal. We have the same goal of maximizing the output data utility in XML streams. However, instead of a simplistic model of certain domain value denoting utility, we must consider the complexity as well as context of XML structure and XQuery. we do not specify importance based on different value interval. Instead we let user to denote the importance of different patterns in the query.

Most approximate query processing works focus on the max-subset goal, which is, to maximize the output rate [13, 9, 2].[9] provides an optimal offline algorithm for join processing with sliding windows where the tuples that will arrive in future are known to the algorithm. An online algorithm which does not know which tuples will arrive in the future is giving under assumption about certain arrival possibilities. [24] proposes a novel age-based stream model and give the load shedding approach for join processing with sliding windows under memory-limited resources. We can apply their techniques into join processing among multiple XML stream systems if our goal is to get max-subset instead of maximizing output utility. In addition, we are considering looking for shed query for XML streams under CPU limited scenario. For CPU limitation scenario, [13] provides an adaptive CPU load shedding approach for window stream joins in relational stream systems. It follows a selective processing methodology by keeping tuples within the windows, but processing them against a subset of the tuples in the opposite window.[1] also discussed how to perform shedding under CPU limited scenario. They proposes an

$Insert-No-Probe$ and $Probe-No-Insert$ shedding approach which is limited to window join in relational streams.

We can divide approximate query processing work into the two categories based on the query execution paradigm. One type is to keep the query unchanged and instead try to utilize available system resources efficiently to maximize the output, which is a subset of output which would have been generate without resource limitations. The other reduces workload by changing query explicitly. [21] mentioned changing query at operator level. This is similar to our removal some patterns from the query. However, our goal is to maximize output utility instead of maximizing output rate.

Preference model is a natural way for decision making purpose. It is used in many applications, such as e-commerce and personalized web services. [18] proposes Preference SQL, an extension language SQL which is able to support user-definable preference for personalized search engines. It supports some basic preference types, like approximation, maximization and favorites preference, as well as complex preference. Preference XPath [26] provides a language to help users in E-commerce to express explicit preference in the form of XPath query. For view synchronization in dynamic distributed environments, EVE[19] proposes E-SQL, an extended view definition language by which view definer can embed their preferences about view evolution into the view definition. However, their preference model is different with ours:

9. Experiment Results

We use ToXgene[5], an XML data generator, to generate XML documents. All the experiments are run on a 2.8GHz Pentium processor with 512MB memory. We perform three sets of experiments. The first one shows that when we run query on different utility settings, the output utility for greedy and exhaustive are better than random shedding approach. The second set of experiments examine the possible factors to affect output data utility. It shows that different preference model and the pattern sizes would impact the output utility. The third set of experiments compares the overhead of three shedding strategies. It shows the greedy shedding approach has little overhead which is similar as random shedding. However, the overhead of exhaustive is big when the query size scales. The final set of experiments shows with random assignment of preference, greedy can achieve close-to-maximum output utility compared to dynamic programming approach.

9.1. Effect of Arrival Rate

In this set of experiments, we study the output utility variation with varying arrival rate using different shedding approaches. We use query Q1 as running query. The performance is measured by output utility by checking the output utility per second. Once the structural join is performed, the joined tuples are purged from the buffer. Fig. 11 shows the output data utility for query Q1 using exhaustive approach and greedy approach is about 20% higher than that of random approach.

9.2. Effect of Preference Assignment and Pattern Size

The second set of experiments show that the output utility is affected by the assignment of preferences as well as the size of patterns in the query. It also implies that the assignment of preference affect which shed query will be chosen to run at shedding phase. The definition of pattern size is given by:

$$
\begin{aligned}
&Size\ of\ Pattern\ P_i \\
&= \{average\ number\ of\ P_i\ elements\ per\ topmost \\
&element\} * \{average\ number\ of\ tokens\ in\ an\ element\}
\end{aligned}
$$

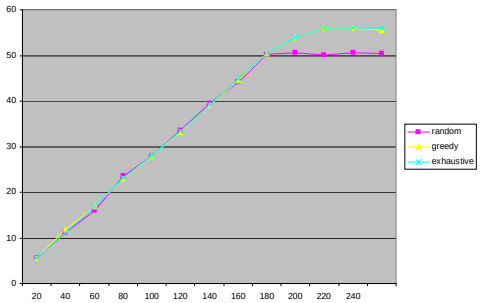

Figure 11. The output utility change with varying arrival rates for three shedding strategies

Note that the exhaustive and greedy approach tend to choose the query which has higher utility with low cost. In this set of experiments, we use a different query Q3.

```
Q1:
$o in document("a.xml")/list/transaction
  return <result>
      $o/category, $o/addr, $o/item/price,
      $o/item/name, $o/item/description
          </result>
```

Q3 tries to find for every transaction element, its category, shipping address, prices of each item, name of each item and description for each item.

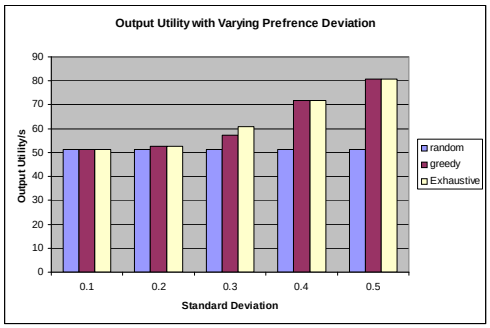

Figure 12. Data Utility Difference Between Random, Greedy and Exhaustive Shedding Strategies with Varying Assignment of Preference Model

Figure 12 shows that the output utility is higher when there is bigger variance among preference values when each pattern in query has same size. Observe that when the difference among preference values is very small, there is little difference for the output utility for three approaches. However, the difference of output utility would be different when the standard deviation of preference values reaches 0.5. Figure 13 shows the output utility changes with varying pattern sizes given the same time period. Here all the patterns(assume they are independent) in the query are independent and of equal preference. Observe that the output utility for data with greater standard deviation using random approach is decreased because such data require higher processing cost than the data with smaller standard deviation. However, this is not the case for greedy and exhaustive approach. The output utility

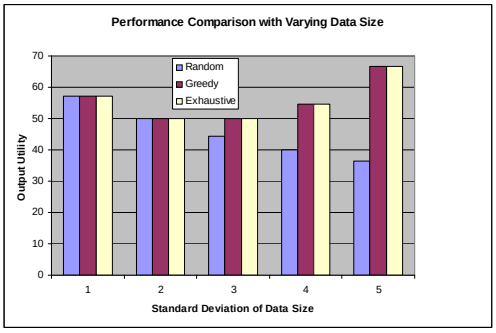

Figure 13. Data Utility Difference Between Random, Greedy and Exhaustive Shedding Strategies with Varying Pattern Size

for these two approaches is much higher than random approach when the size deviation size is 5. This is because the query with small size patterns have smaller location cost and buffering cost, which result in lower overall processing cost. In this case greedy and exhaustive shedding approach would pick such shed query since they have relative higher utility.

9.3. Overhead of Shedding Approaches

In this section, we study the overhead of three shedding strategies. The overhead of shedding approach is measured by the time spent on choosing which shed query to run at shedding phase. We study whether the more complex query is, the overhead is increased dramatically. We use five queries which vary on number of patterns. From the figure shown in Figure 14, we can investigate even when the query become more complex, the overhead of greedy approach is still very small, although it is a bit higher than random shedding. But it does not scale when the query becomes more complex. However, for exhaustive approach, it is already very high when the number of patterns in query is 5. This would cause the total data utility for a certain time period decrease. Thus the overhead of exhaustive is very big and not desirable.

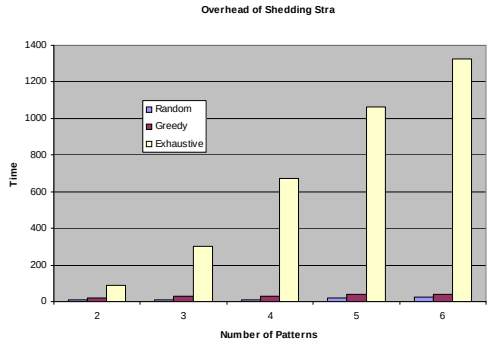

Figure 14. Overhead difference between random, greedy and exhaustive shedding approach

9.4. Random Experiments on Three Shedding Approaches

In the first and second set of experiments, we can observe the greedy and exhaustive approach perform better than random shedding approach on output utility. However, we only compare them based on limited number of preference settings. People would be concerned about which approach can generate better results for most cases, i.e. What is the general comparison result of these shedding approaches on output utility. In order to compare these three shedding approaches, we generate 1000 sets of preference model which satisfy our constraints on preference model. Then we compare greedy versus exhaustive and random versus greedy separately. We run experiments on these 1000 sets of sample data and compare their output utility. Figure 15 shows the histogram on utility ratio of output utility of greedy over exhaustive approach. We can observe that the output utility ratio of greedy approach over exhaustive approach is skewed left. The height of the bar where the ratio equal to 1 is the highest. In addition, about 80 percent data stay in the area where the ratio is over 0.8. This means that greedy can get close to optimal result in most cases. Figure 16 shows that the histogram on output utility ratio of random over greedy approach. Observe that the utility ratio of random over greedy approach is skewed right. Most data are staying in the area where the ratio of random over greedy is less than 0.6. Only very few percent of data can reach the ratio 1 which means random approach has same output utility as greedy approach.

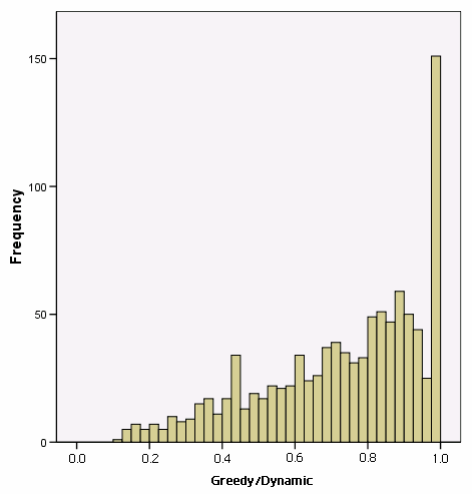

Figure 15. Histogram of output utility Ratio of Greedy over Exhaustive Approach

10 Acknowledgments

References

[1] A. Ayad, J. Naughton, S. Wright, and U. Srivastava. Approximating streaming window joins under cpu limitations. Technical report, 2005.

[2] A. M. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD*, 2004.

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

[4] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *ICDE*, pages 350–361, 2004.

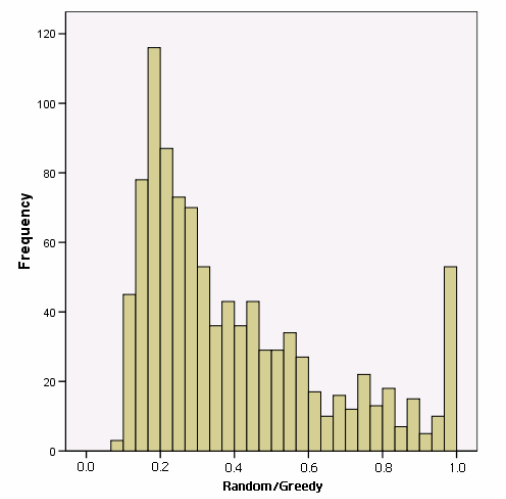

Figure 16. Histogram of output utility Ratio of of Random over Greedy Approach

[5] 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.

[6] 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)*, pages 228–239, 2004.

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

[8] J. Chomicki. Preference formulas in relational queries. *ACM Trans. Database Syst.*, 28(4):427–466, 2003.

[9] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD*, pages 40–51, 2003.

[10] Y. Diao and M. Franklin. Query Processing for High-Volume XML Message Brokering. In *International Conference on Very Large Data Bases (VLDB)*, pages 261–272, 2003.

[11] P. C. Fishburn. Utility theory for decision making. 1970.

[12] P. C. Fishburn. Preference structures and their numerical representations. *Theor. Comput. Sci.*, 217(2):359–383, 1999.

[13] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Adaptive load shedding for windowed stream joins. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 171–178, New York, NY, USA, 2005. ACM Press.

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

[15] H. Su, E. A. Rundensteiner and M. Mani. Raindrop: An XQuery Engine over XML Streams - on Semantic Query Optimization (demonstration). In *VLDB*, 2004.

[16] Z. Ives, A. Halevy, and D. Weld. An XML Query Engine for Network-Bound Data. *VLDB Journal*, 11 (4): 380–402, 2002.

[17] W. Kießling and H. B. Optimizing preference queries for personalized web services. In *IASTED International Conference on Communications, Internet and Information Technology*, 2002.

[18] W. Kießling and G. Köstler. Preference sql - design, implementation, experiences. In *VLDB*, pages 990–1001, 2002.

[19] A. J. Lee, A. Nica, and E. A. Rundensteiner. The eve approach: View synchronization in dynamic distributed environments. *IEEE Trans. Knowl. Data Eng.*, 14(5):931–954, 2002.

[20] B. Ludascher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *International Conference on Very Large Data Bases (VLDB)*, pages 227–238, 2002.

[21] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

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

[23] D. Pisinge. *Algorithms for Knapsack Problem*. PhD thesis, University of Copenhagen, 1995.

[24] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *VLDB*, pages 324–335, 2004.

[25] N. Tatbul, U. Çetintemel, and et. al. Load shedding on data streams. In *VLDB*, 2003.

[26] H. B. Werner Kießling and F. S. Preference xpath- a query language for e-commerce. In *5th International Conference Wirtschaftsinformatic*, pages 43–62, Augsburg, Germany, 2001.