Utility-driven Load Shedding for XML Stream Processing *

Mingzhu Wei , Elke A. Rundensteiner and Murali Mani
Department of Computer Science
Worcester Polytechnic Institute, USA
{samanwei|rundenst|mmani}@cs.wpi.edu

ABSTRACT

Because of the high volume and unpredictable arrival rate, stream processing systems may not always be able to keep up with the input data streams— resulting in buffer overflow and uncontrolled loss of data. Load shedding, the prevalent strategy for solving this overflow problem, has so far only been considered for relational stream processing, but not for XML. Shedding applied to XML stream processing brings new opportunities and challenges due to complex nested nature of XML structures. In this paper, we tackle this unsolved XML shedding problem using a three-pronged approach. First, we develop an XQuery preference model that enables users to specify the relative importance of preserving different subpatterns in the XML result structure. This transforms shedding into the problem of rewriting the user query into shed queries that return approximate query answers with utility as measured by the given user preference model. Second, we develop a cost model to compare the performance of alternate shed queries. Third, we develop two shedding algorithms, OptShed and FastShed. OptShed guarantees to find an optimal solution however at the cost of exponential complexity. FastShed, as confirmed by our experiments, achieves a close-to-optimal result in a wide range of test cases. Finally we describe the in-automaton shedding mechanism for XQuery stream engines. The experiments show that our proposed utility-driven shedding solutions consistently achieve higher utility results compared to the existing relational shedding techniques.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems-*query processing*

General Terms

Algorithms, Design

Keywords

XML streams, XML query processing, load shedding, preference model

1. INTRODUCTION

XML has been widely accepted as the standard data representation for information exchange on the web. XML stream systems have particularly attracted researcher's interest recently [5, 8, 14, 21, 16, 22] because of the wide range of potential applications such as online auction, publish/subscribe systems and e-commerce applications. 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 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 monitoring applications, immediate online results often are required, yet system resources tend to be limited given high arrival rate data streams. Therefore, sufficient memory resources may not be available to hold all incoming data and the CPU processing capacity may not be adequate to always handle the stream workload. A common technique to avoid these limitations is load shedding, which drops some data from the input to reduce the memory and CPU requirements of workload. The current state-of-the-art in load shedding for relational stream systems can be categorized into two main approaches [10, 2, 9, 7]. One is random load shedding [10], where a certain percentage of randomly selected tuples is discarded. The other approach is semantic load shedding which assigns priorities to tuples based on their utility to the output application and then sheds those with low priority first.

No work so far discussed load shedding on XML streams where the query results are composed of possibly complex nested structures. Elements, extracted from different positions of XML tree structure, may vary in their importance (utility). Further, these subelements may consume rather different amount of buffer space and require different CPU resources for their extraction, buffering, filtering and assembly. This provides a new opportunity for selectively *shedding XML subelements* to achieve high processing speed. In this paper, we address shedding in the XML stream context and incorporate the "structural utility" for XML elements into the shedding decision.

Consider an online-store, customers may have periods of heavy usage during promotions time or on holidays. The online store may receive huge numbers of order. When the processing capacity is not sufficient to keep up with data arrival rate, the data will accumulate in the buffer resulting in an overflow. In this case, we have to either drop some data or improve the processing speed. We consider the topmost "transaction" element a basic unit based on which we can generate results. However, dropping complete "transaction" elements means that we may lose important information. In this scenario, dropping unimportant but resource-intensive subelements may be more meaningful to applications compared to the complete-tuple-granularity shedding. We call this type of "el-

*This work has been partially supported by the National Science Foundation under Grant No. NSF IIS-0414567 and NSF CNS-0551584.

WWW 2008, April 21–25, 2008, Beijing, China.
ACM 978-1-60558-085-2/08/04.

ement" granularity drop *structural shedding* since it changes the structure of query results.

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

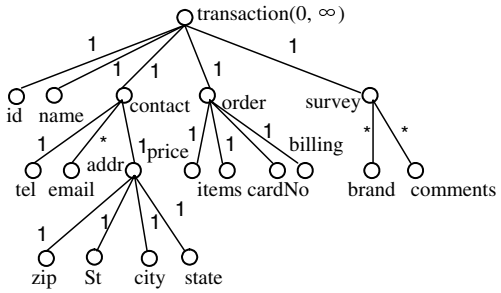

Figure 1: The schema definition for Q1

Let us consider the query Q1 issued above. This query returns the item list and contact information including telephone, email and address when customers spend more than 100 dollars. To process as many transaction elements as possible, consumers of the query result may prefer to selectively obtain partial yet important content as 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 shown in the schema in Figure 1. This means we process more tokens for each single "addr" element; 2) "addr" element may be "optional" to output consumers 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" elements from the input tokens. In this case, we bypass the processing of tokens from "<addr>" to "</addr>". Second, we no longer need to buffer "addr" element during processing. Thus the buffering costs for "addr" element are saved. Note here this shedding can be achieved by removing the "addr" element from the initial query. We call the new reduced query *shed query*.

There are many options to drop subelements from a given query. However, different shed queries vary in their importance and their processing costs. Hence the correct choice of appropriate shed queries raises many challenges. First, what model do we employ to specify the importance of each subelement? Second, after generating different shed queries, how can we estimate the cost of these shed queries at runtime? Third, which of the potential shed queries should be chosen to obtain maximum output utility? Our solution tackles these challenges using a three-pronged strategy. One, we propose a preference model for XQuery to enable output consumers to specify the relative *utility* (a.k.a preference) of preserving different sub-patterns in the query. Two, we develop a cost model to estimate the processing cost for the candidate shed queries. Three, we transform the shed query decision problem into an optimization problem. The main goal of our shedding technique is to maximize output utility given the stream input rate and limited computational resources. Our contributions are summarized as below:

1. In this paper, we define a structure-based preference model which uniquely exploits the relative importance of different sub-patterns in XML query results.
2. We formulate the shedding problem as an optimization problem to find the shed queries that maximize the output utility based on our structure-based preference model and the estimated cost derived from our cost model for XML streams.
3. To solve the shedding problem, we develop two classes of algorithms, OptShed and FastShed. OptShed guarantees to find an optimal solution however at the cost of an exponential complexity. FastShed achieves a close-to-optimal result in a wide range of cases.
4. We propose a simple yet elegant in-automaton shedding mechanism by suspending the appropriate states in the automaton-based execution engine, in order to drop data early (and thus efficiently).
5. We provide a thorough experimental evaluation that demonstrates that our approach maximizes the utility while keeping CPU costs under the system capacity.

The remainder of this paper is organized as follows: In Section 2 we describe the query pattern trees and how to generate different shed queries for a given query. Section 3 describes the cost model for XML stream processing. In Section 4, we define how preferences can be specified for different query patterns and how to compute the preferences for queries. In Section 5, we formulate the shedding problem and provide two algorithms. Implementation of our in-automaton shedding strategy is described in Section 6 while the experimental results are shown in Section 7. Section 8 discusses the related work and conclusions are given in Section 9.

2. PRELIMINARIES

2.1 Query Pattern Tree

We support the core subset of XQuery in the form of "FOR WHERE RETURN" expressions (referred to as FWR) where the "RETURN" clause can contain further FWR expressions; and the "WHERE" clause contains conjunctive selection predicates, each predicate being an operation between a variable and a constant. We assume the queries have been normalized as in [6].

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 use single line edges to denote the parent-children relationship or attributes and double line edges to denote the ancestor-descendant relationship.

We define the following terms in an XQuery. First, a *context variable* corresponds to an XPath in the "FOR" clause, e.g., *$a* in Figure 2. Context variables must evaluate to a non-empty set of bindings for the FWR expression to return any result. Second, a pattern that corresponds to an XPath in the "RETURN" clause, e.g., $*a*/*contact*/*tel* or $*a*//*name*, is called *return pattern* ("r" pattern). Return patterns are optional, meaning even if $*a*/*contact*/*tel* evaluates to be empty, other elements will still be constructed. Third, a *selection pattern* ("s" pattern) corresponds to an XPath in the "WHERE" clause, i.e. it has associated predicates. For instance, the XPath, $*a*/order/*price* in Figure 2 is a selection pattern. The context variable, "r" and "s" patterns for query Q1 are annotated on their destination nodes in Figure 2. We call the destination nodes of the return and selection patterns "r" and "s" nodes respectively.

2.2 Generating Shed Queries

We now investigate how to generate shed queries based on an original query. We distinguish between two terms, sub query and shed query. Sub queries are generated by removing one or multiple

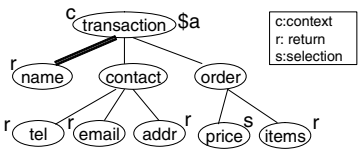

Figure 2: Query Pattern Tree for Q1

nodes from the initial query tree. A shed query is a valid sub query, and it obeys the following rules:

1. A shed query always has the same root as the initial query.
2. The leaf nodes of a shed query have to be either "r" or "s" nodes.

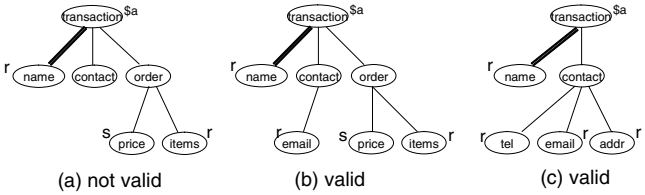

Figure 3: Shed Query Trees

For instance, Figure 3(a) is not valid because this tree does not need to keep the "contact" element because all children of the "contact" element are removed and the XPath $\$a/contact$ is neither an "r" nor an "s" pattern. Figures 3(b) and (c) show two valid sub queries for query Q1.

Assume B denotes the number of all "r" and "s" patterns for a given query tree. When the query tree is a completely flat tree of height 1 and width B, the maximum number of shed queries is 2^B. When the query tree is deep and has only one node on each level, at most B shed queries exist. Thus the number of shed queries for a query varies between B and 2^B.

3. COST MODEL OF XML STREAM SYSTEMS

3.1 Automaton Processing Model

As is known, automata are widely used for pattern retrieval over XML token streams [8, 21, 15]. The relevant tokens are assembled into elements to be further filtered or returned as final output elements. The formed elements are then passed up to perform structural join and filtering. An algebra plan located on top of the automaton for query Q1 is shown in Figure 4. An Extract operator is responsible for collecting tokens for some pattern and composing them into XML elements. For instance, $Extract\$a//name$ collects tokens to form "name" elements. Structural join operator is responsible for combining the elements from its branch operators based on structural relationship and form a transaction tuple. Observe that the context variable $\$a$ in the "FOR" clause is mapped to a structural join. In addition we perform selection on $a/order/price to judge whether the "price" is greater than 100. Thus we have the

Notation	Explanation
N^{P_i}	Number of elements matching P_i for a topmost element
n_{start}	Total number of start or end tags for a topmost element
S^{P_i}	Number of tokens contained in an element matching P_i
A	Set of states in automaton
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 processing a start tag of an element in the query
C_{null}	cost of processing a start tag of an element not in the query
$C_{backtrack}$	cost of popping off states at the stack top
C_{buf}	cost of buffering a token

Table 1: Notations Used in Cost Model

following query processing tasks in XML stream systems: 1. Using automaton to locate tokens. 2. Extracting tokens. 3. Manipulating buffered data, which includes structural join and selection.

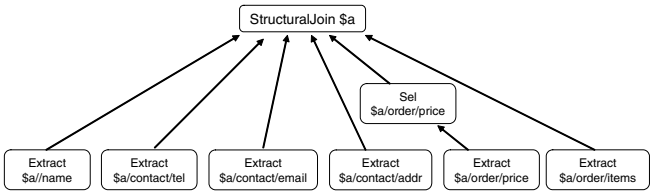

Figure 4: An Example Plan

3.2 CPU Cost Model for a Query

We now design a cost model to estimate the processing costs of shed queries for XML streams. This cost model is adapted from the cost model proposed in [25]. In XML streams we measure the query cost for a complete topmost element since it is the basic unit based on which we generate query results. We call the processing time of handling such a topmost element the *Unit Processing Cost* (UPC). For instance, the cost of query Q1 thus is the unit processing cost of handling one "transaction" element.

We divide the UPC for XQuery into three parts: *Unit Locating 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. UPC is equal to the sum of the cost of these three parts. When we drop either "r" patterns or "s" patterns from the query, we estimate the cost change for these three parts. Note that for a new shed query, its processing cost might not be reduced when dropping "s" patterns. Although it appears that the evaluation cost of the selection pattern is saved, it might need to construct more nodes. In this case the UPC might even be increased if the selectivity of the "s" pattern is not 1. Due to space limitations, we only discuss ULC and UBC here. UMC and the discussion about the selectivity of "s" patterns can be seen in [27].

Unit Locating Cost (ULC). In locating tokens, when an incoming token is a start tag, we need to check whether this start tag will lead to any transitions. If it is transitioned to a new state, tasks to be undertaken may include setting a flag to henceforth buffer tokens or to record the start of a pattern. We call such a transition cost $C_{transit}$. The start tokens of all elements in the query tree will cause such a transition. When there are no states to transition to, an empty state is instead pushed onto the stack top. All start tokens of patterns that do not appear in the query tree will lead to such an empty state transition. The cost associated with this case is

C_{null}. For instance, when $< id >$ is encountered, an empty state is pushed onto the stack top. When the incoming token is an end tag, the automaton pops off the states at the top of the stack. We refer to such popping off cost as $C_{backtrack}$. The popping costs for all end tags are the same. The relevant notations are given in Table 1.

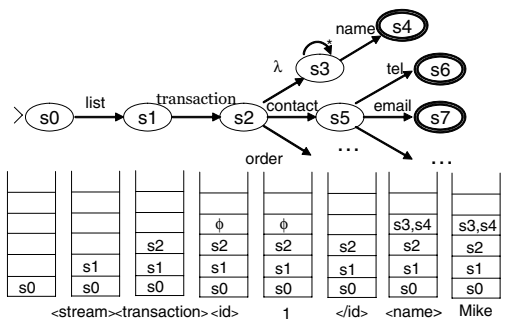

Figure 5: Snapshots of Automaton Stack

We split ULC into two parts, one considers the cost of locating the start and end tags for elements in the query tree, and the other considers the cost for locating the start and end tags for other elements. The first part can be measured by the invocation times for each state and the transition cost for a token as below:

$$\sum_{q \in A} n_{active}(q)(C_{transit} + C_{backtrack}) \quad (1)$$

$\sum_{q \in A} n_{active}(q)$ denotes the number of start tags for which non-empty transition exists in automaton. The number of other start tags, namely for elements which are not in the query tree, can be written as $n_{start} - \sum_{q \in A} n_{active}(q)$. Thus the second part of the transition cost is as below:

$$(n_{start} - \sum_{q \in A} n_{active}(q))(C_{null} + C_{backtrack}) \quad (2)$$

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

$$\sum_{q \in A^{P_i}} n_{active}(q)(C_{transit} + C_{backtrack}) \quad (3)$$

However, in the shed query, since these states are never reached, they are now treated as elements that are not in the query. Their locating cost is thus changed to:

$$\sum_{q \in A^{P_i}} n_{active}(q)(C_{null} + C_{backtrack}) \quad (4)$$

Thus Eq(3)- Eq(4) indicate the savings in locating costs gained by switching from the initial query to this shed query Q_s.

Unit Buffering Cost (UBC) In our query engine, we only store those tokens that are required for the further processing of the query. As we mentioned, the Extract operators are responsible for buffering those tokens. Thus each "r" and "s" pattern has a corresponding Extract operator. Such buffering cost for a topmost element is defined as UBC (Unit Buffering Cost). Extract operators are invoked when the corresponding states are reached in the automaton. For example, in Figure 5, state $s4$ would invoke an Extract operator to store the whole "name" element. In addition we assume here the buffering cost is the same for all individual tokens.

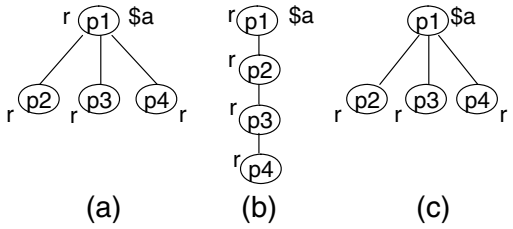

Figure 6: Buffer Sharing Examples

Our buffer manager uses pointers to refer to elements. Thus we do not store the same token more than once. Three query examples are shown in Figure 6. In Figures 6(a) and 6(b), the parent pattern and its children patterns 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 $p2$, $p3$ and $p4$ pointing to $p1$. In this case, the buffering cost is equal to the buffering cost of the parent pattern $p1$. However, in Figure 6(c), since the parent is not an "r" pattern, only its children are to be returned. The buffering cost is equal to the buffering cost of all the children. Hence, for a given query, we need to find all non-overlapping topmost patterns which are either "r" patterns or "s" patterns, called henceforth the *storing pattern set*. The storing pattern set can be obtained by traversing the query tree in a breadth-first manner [27].

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

$$UBC(Q) = \sum_{p \in R} N^p S^p C_{buf} \quad (5)$$

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_i}, n_{start} and $n_{active}(q)$. We collect s^{P_i} in Extract operators. We then use these statistics to estimate the cost of shed queries using the formulas given above. Note that some cost parameters in Table 1 such as $C_{transit}$, C_{null} and C_{buf} are constants. We do not need to measure them during the query execution.

4. PREFERENCE MODEL FOR QUERIES

Value-based Preferences v.s. Structure-based Preferences. In many practical applications, some results are considered more important than other output tuples. For instance, the user might be interested in red cars when buying new cars. In this case the utility of the tuple whose color attribute is equal to "red" is higher than those of the tuples whose colors are not "red". Aurora first considered such value-based preference as part of the QoS requirement and proposed semantic load shedding techniques [10] to maximize output utility. In this case, semantic load shedding is achieved by adopting a value-based filter. We can easily incorporate such value-based preference and their filter-based shedding approach in the XML stream scenario. However, this is not our main interest in this XML work. Instead, we are interested in exploring the structure-based preference in XML stream processing. In the XML stream scenario, the input stream as well as the output results are composed of different XML subelements instead of just flat attributes,

and hence more complex than relational tuples. The importance of different elements in an XML tree may vary due to their semantics. As illustrated via an example in Section 1, the "email" element is considered more important than the "addr" element as "email" is a faster and more convenient means to notify customers.

Specifying Preferences in Query. We distinguish between two options to specify preferences, one is to specify preferences in the data schema and then derive the preferences for the patterns in the query, and the other is to specify preferences directly in the query. The former case is somewhat rigid when the same data is consumed by different applications. For instance, given store sale data, the data mining expert would think the customers' information including gender, age, education and their shopping lists are important since they want to learn about the correlation between customers' background with their shopping interests. However, the stock manager would be interested in the products and their sale quantity. In this case, users may assign preferences rather differently to the same subelements. Thus having a single fixed preference on data schema is an unnecessary restriction. For this reason, we instead propose that users specify preferences to the patterns in the query.

To support this, we need a metric to measure the importance of each pattern for a given query. We define a quantitative preference model that represents preferences of preserving different elements in the query result.The preferences can be specified by the user who issues the query or the consumer of the query result. By binding different patterns with their corresponding preferences, shed queries vary in their perceived utilities to the user. In our preference model, we do not distinguish utility assignment of "r" and "s" patterns. Instead, users decide their utilities. However, the differences among processing costs for "r" and "s" patterns are handled by the cost model.

We support two alternative types of preference specification on query patterns. One uses prioritized preference [18] to qualitatively express the relative ranking among different patterns, and the other uses a quantitative approach [13, 12] that directly scores the importance of the patterns. Users are free to choose either the Numerical Preference Model (NPM) or the Prioritized Preference Model (PPM) to represent their preferences on query patterns. For preferences specified by PPM, we translate the prioritized preferences to numerical forms using a score formula. In both cases we use the quantitative metric to compute the utilities of shed queries.

4.1 Numerical Preference Model (NPM)

If a user chooses to specify preferences using NPM, he or she can assign customized utilities (preferences) for different patterns in the query in a numerical form. Users only need to specify the utility values for the "r" and "s" patterns. The utility of pattern P_i where P_i is a "r" or "s" pattern is represented below:

$$\nu(P_j) \mapsto [0, 1]$$

Here $\nu(P_j)$ is a value between [0,1]. An example of utility assignment for query Q1 is shown in Figure 7 (destination node of each pattern is annotated by its utility value).

4.2 Prioritized Preference Model (PPM)

If users choose to use the prioritized preferences, they describe the relationship among patterns. This means that given a query, the user declares the relative ordering of "r" and "s" patterns in term of their importance. Note that we do not require users to specify the preference ordering for all the patterns since users may only specify the ordering for some patterns. An example prioritized preference for query Q1 is:

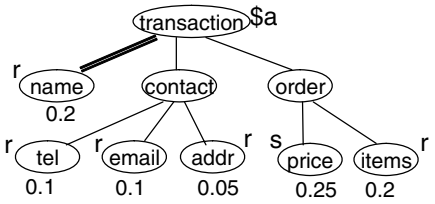

Figure 7: Q1 Query Tree with Preferences

$\$a//name \succ \$a/order/price \succ \$a/contact/tel \succ \$a/order/items \succ \$a/contact/email \succ \$a/contact/addr$

For the above qualitative preference representations, we translate them to quantitative preferences. A score assignment strategy is applied based on the given prioritized preference ranking, where we assign scores using the following formula:

$$\nu(Pattern\ Ranking\ k) = 1/2^k$$

For instance, the utility for pattern $\$a//name$ is equal to $\frac{1}{2}$ and the utility for $\$a/contact/tel$ is equal to $\frac{1}{2^3}$. The reason why the preference of pattern ranking k is translated to$\frac{1}{2^k}$ is explained in Section 4.3. When only the ordering of some patterns is specified, the scoring scheme below will also generate the preferences for those patterns that are not ranked.

4.3 Scoring Scheme for Patterns without Preferences

We do not require users to specify the preferences for all the "r" and "s" patterns. In this case we obtain the utilities for those patterns using the following properties:

1. *Precedent parent*: A parent pattern is more important than its descendant patterns. This is because parent return nodes always contain all the descendant "r" and "s" patterns. For a non-leaf pattern that has not been assigned preferences, its utility is defined as the sum of scores of all its children.

2. *Equivalent leaf*: We assume the leaf nodes without assigned preferences are equally important. Their preference values are thus the same. They are less important than the patterns who have been assigned preferences. Let w denote the number of patterns that are not assigned preferences, their utilities are all assigned to

$$min(\nu(P_j)) * 1/2w\ ,$$

where $min(\nu(P_j))$ is the minimum value among all assigned preferences.

Now we observe that the translation formula for prioritized preference model can guarantee the precedent parent property if the user specifies the pattern is more important than any of its descendants.

4.4 Computing Utilities for Queries

After the quantitative preferences for all the patterns in the query are determined, we can calculate the utility of the original query and the shed queries derived from the original query. If a pattern

appears in a query tree of a shed query, that means it will be considered into the query and its utility is obtained. We use utility of a query to indicate the amount of utility users gain by executing this particular query Q on a single topmost element, in other word, how much utility is obtained by including all the patterns in this shed query. It can be calculated as

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

Where P_j is either a "r" pattern or "s" pattern. For instance, the utility of Q1 is: 0.2+0.1+0.1+0.25+0.2+0.05= 0.9.

Particularly, we introduce the empty query, a special shed query which actually drops the whole topmost element. For the empty query Q_0, we define its utility $\nu(Q_0) = 0$ since it does not contribute to any output.

After calculating the preference for a given query, we perform a simple *normalization* process. Assume the preference for a shed query is $\nu(Q_i)$ and the preference for the original query is $\nu(Q)$. The preferences for each shed query is normalized to $\nu(Q_i)/\nu(Q)$ and the preference for the original query is 1. After the normalization, we can observe that the normalized preferences of the shed queries including original query and empty query would fall into [0, 1]. Note that in the later sections, we use normalized utility values for the shed queries.

[11] proposed an extension of XPath which incorporates value-based preferences into XPath. Similarly we can easily extend the XQuery syntax to integrate our structure-based preferences into an XQuery expression as below:

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

5. SHEDDING ALGORITHMS

5.1 Decide When to Shed

The problem of deciding when the system needs to shed input data has been discussed in other works [10]. This is not specific to XML stream systems. In our system we adopt the following approach for simplicity. We assume a fixed memory to buffer the incoming XML stream data. As soon as all tokens in an XML element have been processed, we clean those tokens from the buffer. We assume a threshold on the memory buffer that allows us to endure periodic spikes of the input without causing any overflow. During execution, we monitor the current memory buffer. When buffer occupancy exceeds the threshold, we trigger the shedding algorithm.

5.2 Formulation of Shedding Problem

Let us assume that the shed query set is $\{Q_0, Q_1, ..Q_n\}$ where Q_0 is the empty query and Q_1 is the original query. Here empty query just drops all the tokens of a topmost element. The reason why we introduce empty query Q_0 into shed query set is for the convenience of the formalization of the shedding problem, so that all the input elements are consumed by shed queries. Since this empty query does not generate any output, we assume the utility of empty query Q_0 denoted by ν_0 and the UPC of Q_0 denoted by C_0 are both zero. The goal of the shedding problem is to find which shed queries will be chosen to run in order to achieve maximum utility. We have the following inputs to our shedding problem: 1. data arrival rate λ in the unit of topmost elements per time unit; 2. utilities of candidates in the query set $\{\nu_0, \nu_1, ..\nu_n\}$); 3. processing costs (in time units) of queries in the set $\{C_0, C_1, ..C_n\}$. 4. the number of time units for shedding query to execute, C, denoting the available CPU resources.

We aim to find a set of shed queries which satisfies the two conditions: (1) Consume all the input elements in C time units. Here C is an integer to measure CPU resources.(2) Maximize the output utility. Note that the shed queries here include empty query, original query and shed queries we derived from original query. We could consider variation of the problem by imposing additional constraints. If we limit the number of qualified queries in the result set to only one, we have to check all the shed queries to see whether any shed query can consume all the input elements. If there exists such shed queries, we would pick the query that yields the highest utility. However, it is possible that all the shed queries except the empty query are too slow to be able to consume all the inputs. In this case, the empty query is the only option since it can consume all the inputs. Unfortunately, the output utility would be zero since we drop everything. Thus restricting to one query is not sufficient to achieve optimal results.

Another option is to restrict the number of shed queries to two. As mentioned before, there might not exist such a shed query from the query set whose processing speed is as fast as input arrival rate except empty query. It implies that if picking two queries from the shed queries and none of them is the empty query, we cannot handle all input data. Thus picking the empty query is necessary in this case. Given that the empty query cost is zero, we can formulate this problem below:

Given the constraint: $x_i * C_i <= C$, where $1 \leq i \leq n$ and x_i indicates the number of dropped topmost elements for query Q_i.

We want to maximize output utility $x_i * v_i$. The number of elements to drop (corresponding to empty query) is thus equal to $\lambda - x_i$. Note that the current state-of-the-art shedding techniques [3, 10] can be regarded as a special case for allowing two shed queries, as they typically pick the original query and empty query.

However, allowing only two shed queries might not be optimal. Consider the following example. The utility and cost of three shed queries Q_1, Q_2 and Q_3 are shown below.

{(1, 55ms), (0.9, 45ms), (0.6, 30ms)}

Assume the available CPU resource is 80ms and 3 topmost elements arrive during that time period. If we only allow two different shed queries, we have to let 2 elements execute query Q_3 and 1 element execute empty query. The output utility is 0.6*2+0= 1.2. However, note that if we let 1 element execute query Q_2, 1 element execute Q_3 and 1 element execute empty query, the output utility is even higher, 0.9+0.6+0=1.5. We therefore do not limit the number of different shed queries in the result set. Our goal is to find a coefficient vector $\{x_0, x_1, ..x_n\}$ for the shed query set, which maximizes the utility of the total processed elements while keeping the processing cost below the CPU processing capability. Here x_i denotes the number of topmost elements assigned to query Q_i. The problem is formalized below.

1. The total number of XML elements processed (including those processed by empty query) can be calculated as:

$$X(s) = \sum_{i=0}^{n} x_i \quad (6)$$

2. Total execution cost by consuming all the input elements can be represented as

$$C(s) = \sum_{i=0}^{n} x_i * C_i \quad (7)$$

Using the above equations, the shed problem is to maximize the total data utility:

$$\sum_{i=0}^{n} x_i \nu_i \quad (8)$$

Subject to

$$X(s) = C * \lambda$$
$$\texttt{and } C(s) \leq C \quad (9)$$

Note that the cost of all shed queries is measured in time units, thus they are all non-negative integers. We thus conclude that this problem is an instance of the knapsack problem [17]. We propose two solutions for this problem as described below.

5.3 OptShed Approach

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

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

Here $\psi_j(\tilde{c})$ is a sub-problem which uses queries from Q_0 to Q_j and its cost is less than or equal to $\tilde{c}$.

Clearly, $\psi_n(C)$ gives the optimal solution to the original problem we want to solve, where C denotes the total available CPU resources.

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

$$\phi_j(0) = 0 \ , \ 0 \leq j \leq n$$
$$\phi_j(\tilde{c}) = \texttt{max}\left\{\phi_{j-1}(\tilde{c} - kC_j) + k\nu_j \mid 0 \leq k \leq \lfloor \frac{\tilde{c}}{C_j} \rfloor\right\}$$

From the matrix of sub-problems, we can see that we need to repeat the calculation of $\phi(\tilde{c})$ nC times to get the final result, and each calculation can be finished using a max-value searching algorithm, whose time cost is O($\log_2 C$) [23]. Thus the total time complexity is O($nC \log_2 C$).

5.4 FastShed Approach

Since the time complexity of OptShed is prohibitively expensive in practice, we want to find a simple but effective way to solve this problem. We thus propose an efficient greedy algorithm, called FastShed. Observe that load shedding will be invoked when the arrival rate is greater than the processing speed of the original query, meaning $\lambda \geq \frac{1}{C_1}$. When the arrival rate is greater than the processing speed of all the shed queries, we use a ratio-sorting approach. We calculate the ratios of utility over processing cost, ν_i/C_i, for each candidate query Q_i. We sort all queries in terms of these ratios. Assume that the ratios of Q_{i_1}, Q_{i_2},...,Q_{i_n} are in non-increasing order. We assign Q_{i_1} to as many as possible input XML elements as long as it does not exceed our given CPU processing capability, and then assign Q_{i_2} to as many as possible input XML elements according to the remaining CPU processing capability, and so on.

However, if the arrival rate cannot satisfy the condition that it is greater than the processing speeds of all shed queries, i.e., there exists at least one shed query whose processing speed is greater than the arrival rate, the utility over cost ratio sorting approach might be sub-optimal. Let us examine the following example. Assume the arrival rate is 30 topmost elements/s which is equal to 0.03 elements/ms. Assume the utilities and costs of four shed queries Q_1, Q_2, Q_3 and Q_4 are shown below:

{(1, 40ms), (0.9, 25ms), (0.8, 20ms), (0.7, 50ms)}

Assume the CPU resources are limited to 1000ms. If we rank these queries based on their utility by cost ratio, the decreasing order is Q_3, Q_2, Q_1, Q_4. However, if we choose query Q_3 ,the utility it can reach is actually equal to 0.8*30 =24 instead of 0.8 *1000/20 = 40. This is because the number of elements on which we run a shed query cannot exceed the amount of input data. Thus for the shed query whose processing speed is greater than arrival rate, the output utility is limited to its utility * arrival rate. In this case, the output utilities for query Q_1, Q_2, Q_3 and Q_4 are 25, 27, 24 and 14 respectively. Thus query Q_2 is the shed query we should choose since it yields highest utility.

We account for this case by modifying the ratio sorting approach as follows. We define $\gamma_i = \nu_i * min\{\lambda, \frac{1}{C_i}\}$, and the sorting is done based on these γ_is.

Algorithm 1 FastShed

Input: $\lambda, \{\nu_0, \nu_1, ..\nu_n\}, \{C_0, C_1, ..C_n\}, C$
Output: $\{x_0, x_1, ..x_n\}$
void FastShed()
$\gamma_i = \nu_i * min\{\lambda, \frac{1}{C_i}\}$ $(1 \leq i \leq n)$
Sort queries Q_1,Q_2,...,Q_n so that $\gamma_{i_1} \geq \gamma_{i_2} \geq ... \geq \gamma_{i_n}$
$C' \leftarrow C$
$\lambda' \leftarrow C * \lambda$
for $j = 1$ to n **do**
  $x_{i_j} \leftarrow$ min { $\lfloor C'/C_{i_j} \rfloor, \lambda'$ }
  $C' \leftarrow C' - x_{i_j} * C_{i_j}$
  $\lambda' \leftarrow \lambda' - x_{i_j}$
  if $C' \leq 0$ or $\lambda' \leq 0$ **then break**
end for
$x_0 \leftarrow \lambda - \sum_{j=1}^{n} x_j$

The details are described in Algorithm 1. In FastShed, the ratio sorting cost is O($n \log n$) and cost of "for" loop is O(n) respectively. So the total time complexity is O($n \log n$). Normally, $n \ll C$, so FastShed is much faster than OptShed, though FastShed cannot guarantee to find an optimal solution. However, in Section 7, the experiments show that FastShed indeed tends to find a solution very close to the optimal solution for most cases.

6. SHEDDING MECHANISM

In this section, we examine the implementation of different shedding approaches in XML stream systems. For relational stream systems, one common implementation is to insert drop boxes into the plan [10, 1, 3]. However, many XML stream systems use automata to recognize relevant elements on incoming token streams. We can consider two options where the input data can be dropped. One place is when we recognize the tokens using automaton, the other place is after we have formed the elements from extracted tokens. Since dropping them as early as possible can avoid wasted work, we propose to push the shedding directly into the automaton as described below.

6.1 In-Automata Shedding Mechanism

Here we propose to incorporate shedding into the automaton by disabling states. Assume we want to drop patterns $a//name and

$a/contact/tel. Figure 8 shows where to insert drop boxes in the automaton. To drop pattern $a//name, the automaton would temporarily remove the transition from state $s2$ to $s3$. When the start tag of $name$ element arrives, state $s3$ and $s4$ are not reachable. Thus it would not invoke its downstream operator, $Extract\$a//name$. $Extract\$a//name$ will then be labeled with a "dropped" flag. This flag guarantees that the downstream $StructuralJoin\$a$ operator works correctly. Thus when $StructuralJoin\$a$ checks its input operators one by one, if an input operator is labeled with a "dropped" flag, $StructuralJoin\$a$ skips this input.

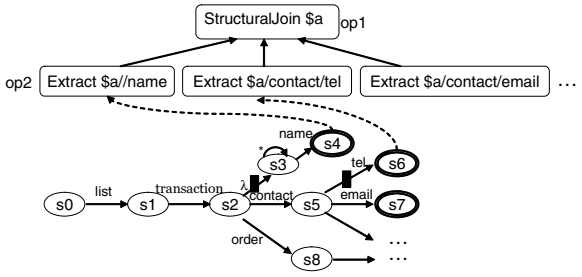

Figure 8: Disable Transition Strategy

6.2 Random Shedding in XML Streams

To compare our shedding solutions with the existing random shedding approach, we have to realize random shedding for XML stream systems. We do not want to disadvantage this existing solution by first storing data in buffer before dropping. Instead we propose to also perform random shedding in the automaton. Since the granularity of incoming data in XML streams is tokens, the start token of the topmost elements is recognized by the automaton. We then can set the "shedding phase" flag to be true. As long as this flag is true, the incoming tokens are dropped. At the same time, we add a drop counter to record how many topmost elements we have dropped. Whenever the end token of the topmost element is identified, the counter's value is increased. If the desired dropping count is reached, the flag is disabled and the system switches back to the "non-shedding" phase.

6.3 Shed Query Switching at Run-time

We support a mixture of shed queries. Assume OptShed provides a solution vector, say <60, 10, 20>. In this case, we will first drop 60 topmost elements, then run query Q_1 for 10 topmost element, then switch to query Q_2 for the next 20 topmost elements. We use a counter to record the number of topmost elements that have been run with query Q_i. After processing the last end tag of the x_ith topmost element, the system restores the removed state transition and then switches to the next shed query. Since the switching happens only after the processing of the last token of the topmost element, it is safe to switch to another query for the next topmost element. Note that here we simply apply the state transition disabling and labeling "dropped" flag, we do not otherwise physically change the plan. Thus the overhead is very small.

7. EXPERIMENTAL RESULTS

We use ToXgene [4] to generate XML documents as our testing data. All experiments are run on a 2.8GHz Pentium processor with 512MB memory. We use query Q1 as testing query and the testing data files are about 30 MB. We perform four sets of experiments. The first one shows that output utility changes with varying arrival rates for all three shedding approaches (Random, OptShed and FastShed). The second set of experiments demonstrates that different distributions of pattern preference settings and pattern sizes impact the output utility. The third set compares the overhead of three shedding strategies. It shows that FastShed has little overhead, similar to Random shedding. However, the overhead of OptShed becomes big for large query sizes. The final set of experiments shows FastShed achieves close-to-maximum utility in practically all cases considered.

7.1 Comparing Three Shedding Approaches

In this set of experiments, we study the output utility changes with varying arrival rates for the three shedding approaches. Fig. 9 shows the output data utility per second for query Q1. Note that in Fig. 9 the three slopes increase the same way when arrival rate is less than 180 topmost elements/s because no shedding happens at that time. After the arrival rate reaches 180 topmost elements/s, the utility of Random remains stable because it has reached its processing capacity. However, FastShed and OptShed achieve higher utility because they choose a shed query which generates higher utility than the Random approach.

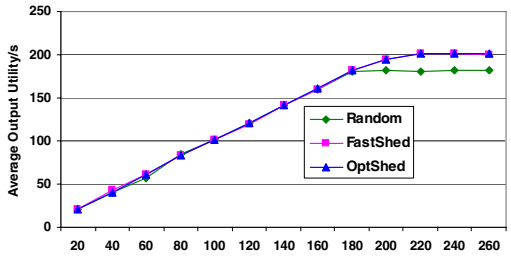

Figure 9: Output Utility Changes with Varying Arrival Rates

7.2 Effect of Preference and Pattern Size

Next, we illustrate the output utility is affected by the distribution of pattern preferences as well as the pattern sizes in the query. It implies that the assignment of preferences indeed affects which shed query will be chosen to run at shedding phase. The definition of pattern size is given by: $P_i = N^{P_i} * S^{P_i}$ where N^{P_i} is the number of elements corresponding to pattern P_i in a topmost element and S^{P_i} is the average number of tokens contained in a P_i element.

We use five different sets of preference settings which differ in their standard deviations. We run query Q1 on the same data set. Each pattern has the same size and each set has the same utility for the initial query. Figure 10 shows that the output utility is higher when there is a bigger variance among pattern preference settings for FastShed and OptShed. We observe that the utilities of the query achieved by the Random approach are the same because the initial query is executed in this case. However, OptShed and FastShed perform differently when the standard deviation for preferences changes. Observe that when the standard deviation of preference values is small, there is little difference among utilities for the three approaches. However, the difference of output utility is significant when the standard deviation of preference values reaches 0.5.

To illustrate the output utility is affected by the pattern sizes, we

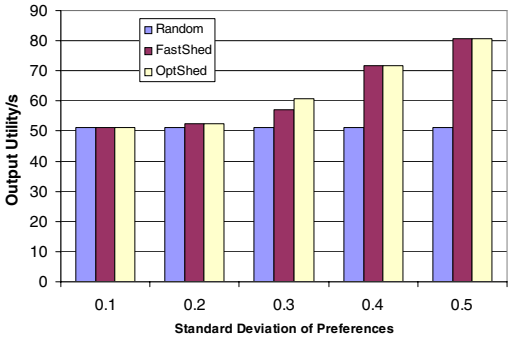

Figure 10: Data Utilities for Varying Preference Assignments

generate five testing data files which differ in their standard deviation of element sizes.We run the query Q2 below.

```
Q2: FOR $o in stream("sample")/list/o
  RETURN $o/P1, $o/P2, $o/P3, $o/P4
```

Note that each data file only contains the elements in the query and the sums of all element sizes in each data file are all equal to 200 tokens. In addition we assume all patterns in the query are independent and of equal preference. Figure 11 shows the output utility changes with varying standard deviation of pattern size during the same time period. Observe that for the Random approach, the output utilities do not change a lot since the UPC of the original query for these four data files are almost the same. However, for FastShed and OptShed, the output utilities are much higher than the utilities achieved by Random approach when the standard deviation of pattern size increases. This is because the shed queries with smaller patterns has smaller locating cost and buffering cost, resulting in lower overall processing cost. In this case FastShed and OptShed would pick such shed queries since they have relatively higher utility/cost ratios and thus higher utilities.

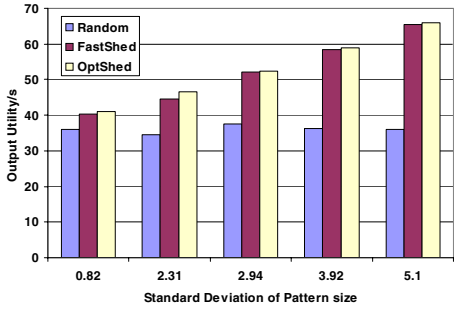

Figure 11: Data Utilities for Varying Pattern Size

7.3 Overhead of Shedding Approaches

Here we study the overhead of the three shedding strategies. The overhead is measured by the time spent on choosing which shed query to run during the shedding phase. We study whether with more complex queries the overhead increases dramatically. We use five queries which vary in the number of patterns. From Figure 12, we observe even when the query becomes complex, the overhead of FastShed is still very small, although it is a bit higher than Random shedding. However, for OptShed, overhead is already very high when the number of patterns in the query is 5. Thus the overhead of OptShed is very big, implying it as an undesirable choice.

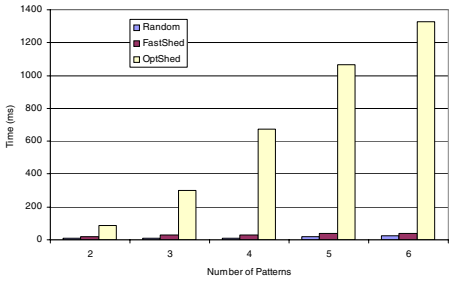

Figure 12: Overhead of Three Shedding Approaches

7.4 Additional Experiments

In the first two experiments above, we observe that FastShed and OptShed perform better than Random shedding on output utility. However, we only compared them based on a limited number of preference settings. Now, we want to study performance of these methods over a wide range of cases. We generate 1000 sets of sample costs and utility measures, where a sample set is generated by assigning preferences to different query patterns randomly. The costs of different shed queries in a sample set are assigned randomly in the range [10, 20], and at the same time ensuring that the cost of a "smaller" query is less than the cost of a "bigger" query. Then we run the three approaches on these 1000 sets of sample data and compare their output utility. Figure 13(a) shows the histogram on the utility ratios of FastShed over OptShed. We observe that these ratios are skewed to the left. About 80% of them are over 0.8. This means that FastShed can get close to optimal results in most cases. Figure 13(b) shows the histogram of output utility ratios of Random over FastShed. Observe that these ratios are skewed to the right. Most of them are less than 0.6. Thus FastShed is much better than Random shedding.

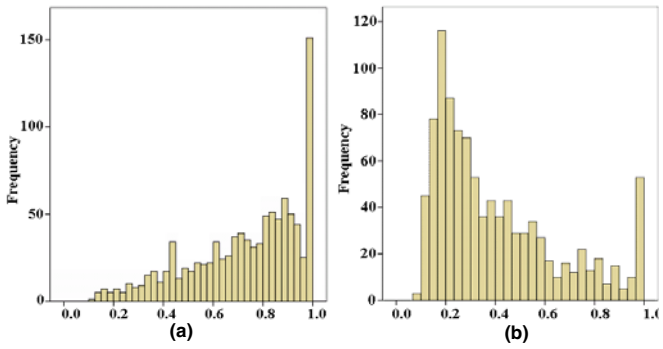

Figure 13: Utility Ratios of (a) FastShed over OptShed (b) Random over FastShed

8. RELATED WORK

In streaming systems, load shedding and sampling data are two common ways to reduce the system workload. Load shedding on relational streaming data has first been proposed in Aurora [10].

This work introduces two types of load shedding: random and semantic. Based on the analysis of the loss/gain rate, random load shedding determines the amount to shed to guarantee the output rate. For semantic drop, they assume that different tuple values vary in terms of utility to the application. In XML streams, instead of a simplistic model of certain domain value denoting utility, we consider the complexity as well as importance of XML result structures in order to make shed query decisions.

Most approximate query processing works in relational streams focus on the max-subset goal, namely, to maximize the output rate [9, 7, 1, 26]. [7] provides an optimal offline algorithm for join processing with sliding windows where the tuples that will arrive in the future are known to the algorithm. [24] proposes an age-based stream model and give the load shedding approach for join processing with sliding windows under memory-limited resources. For CPU limitation scenario, [9] provides an adaptive CPU load shedding approach for window stream joins which follows a selective processing tuple methodology in windows. We can not apply these approximate processing techniques directly into our work since we are targeting a single XML stream without window constraints.

Preference model is used for decision making purposes in many applications, such as e-commerce and personalized web services. As mentioned before, Aurora [10] combines the utility of different tuple values into quality of service. [19] proposes Preference SQL, an extension of SQL which is able to support user-definable preferences for personalized search engines. It supports some basic preference types, like approximation, maximization and favorites preference, as well as complex preferences. Preference XPath [11] provides a language to help users in E-commerce to express explicit preferences in the form of XPath queries. For view synchronization in dynamic distributed environments, EVE[20] proposes E-SQL, an extended view definition language by which preferences about view evolution can be embedded into the view definition.

9. CONCLUSIONS

In this paper, we propose a new utility-driven load shedding strategy that exploits features specific to XML stream processing. Our preference model for XQuery helps users to customize their preferences on different XML result structures. In addition we design a cost model for estimating the costs of different shed queries. We put forward two shed query search solutions, OptShed and FastShed that choose a subset of shed queries to be executed in order to maximize utility. Our experiments illustrate the performance gains of these two approaches in output utility compared with existing shedding solutions. As our future work, we intend to explore the integration of a value-based preference model into our structure-based preference model solution. Another direction is considering additional objectives beyond output quality, such as output rate.

10. REFERENCES

[1] A. M. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD*, pages 419–430, 2004.

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

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

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

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

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

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

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

[9] B. G. et al. Adaptive load shedding for windowed stream joins. In *CIKM*, pages 171–178, 2005.

[10] N. T. et al. Load shedding on data streams. In *VLDB*, pages 309–320, 2003.

[11] W. K. et al. Preference xpath- a query language for e-commerce. In *5th International Conference Wirtschaftsinformatic*, pages 427–440, 2001.

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

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

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

[15] Hong Su, Jinhui Jian and Elke A. Rundensteiner. Raindrop: A Uniform and Layered Algebraic Framework for XQueries on XML Streams. In *CIKM*, pages 279–286, 2003.

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

[17] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer-Verlag, 2005.

[18] W. Kießling and H. B. Optimizing preference queries for personalized web services. In *Communications, Internet, and Information Technology*, 2002.

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

[20] 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.

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

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

[23] D. Pisinger. *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] H. Su, E. A. Rundensteiner, and M. Mani. Automaton in or out: Run-time plan optimization for xml stream processing. In *SSPS Workshop, EDBT*, 2008.

[26] N. Tatbul and S. B. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *VLDB*, pages 799–810, 2006.

[27] M. Wei, E. A. Rundensteiner, and M. Mani. Load shedding in XML streams. Technical report, Worcester Polytechnic Institute, 2007.