Synopsis Based Load Shedding in XML Streams

Ranjan Dash
University of Texas at Arlington
416 Yates Street, PO BOX19015
Arlington, TX 76019
Ranjan.Dash@Yahoo.com

Leonidas Fegaras
University of Texas at Arlington
416 Yates Street, PO BOX19015
Arlington, TX 76019
Fegaras@cse.uta.edu

ABSTRACT

Stream systems are susceptible to variations in data arrival rate. At times, data arrival rate may spike up to cause unacceptable output latencies and unpredictable system behavior. Recently, load shedding systems have been proposed to deal with this situation. But almost all these systems are for relational data streams and, to the best of our knowledge, none has been proposed for XML data streams so far except [15]. Dropping data randomly may have been an effective method for load shedding in the relational context, due to the uniformity of relational data. But in the XML context, the same method will lead to much invasive negative effect on processing of XML queries due to the recursive and nested structure of XML data. We propose a load shedding framework for XML data streams. We explore the effectiveness of various load shedding techniques based on a general load shedding strategy that takes into account QoS parameters and relative accuracy of the query results. We implement various load shedding strategies and present their result.

Categories and Subject Descriptors

H.2.4 [Database Management]: Query Processing

General Terms
Management, Measurement, Performance.

Keywords
Data Streams, XML Streams, Load Shedding, Quality of Service, Synopsis, Histogram, Approximate Query Processing.

1. INTRODUCTION

Some of the major challenges in XML stream processing are: support for near real time processing, handling unpredictable rates of data flow, and limited resources. As the data becomes much varied, undergoes complex processes like joining, these resources fall short of a satisfactory level of qualitative service.

The XML data is usually streamed in the form of SAX events or customized user-defined fragments. Ideally, the processing rate must match the data arrival rate. Most XML query processing systems are typically based on finite state machines or operator pipelines. The demand for CPU grows rapidly with the data arrival rate or the query complexity. When some of the stream sources switch into a bursty mode or wake up from a sleeping mode, the increased demand for CPU outweighs the processing capability that results in buffering more and more events before processing with increasing latency. Long bursts also causes memory overflow. The resolution is to release pressure on critical resources by shedding some of the unprocessed or semi-processed data in a controlled manner, a process known as ‘Load Shedding’. Normally, with load shedding, the query results are approximate, as they lack the contribution from the shedded load. Fortunately, unlike traditional DBMS, stream processing depends on approximations and adaptivity requirements of rapid data streams, which makes load shedding as a viable solution to the overload problem. However, since there is a trade-off between the QoS (Quality of Service) gained by releasing the processing resources and the QoD (Quality of Data) lost by dropping relevant data, our goal is to develop an XML load shedder that is intelligent enough to maximize both QoS and QoD by discriminatingly selecting XML elements to drop based on statistics, but fast enough to catch up with most stream speeds.

The load shedding for relational data streams is implemented extensively in Aurora system [1]. Our structural summary construction is influenced by [2, 6, 7, 11, 12 and 16]; whereas the overall system development is influenced by the feedback approach as explained in [13].

The load shedding problem in XML streams as described in [15] has been addressed more on a cost model using user’s participation to evaluate various shed queries and use them to solve the over loading problem. However our approach is more user transparent. Similarly the work in [15] is more biased towards structure based preferences than value based, where as our work takes into account both structure as well as value properties of complex XML streams to achieve the quality of the result. Our structured predicate load shedding technique strikes middle at both structure and value (predicate) of XML streams. Our processing is different from [15] in another way that their shedding is implemented in automaton which incurs some processing before getting shredded whereas ours is handled much before without any overhead in case of simple random shedding.
In this paper, we formulate a strategy for load shedding in XML streams that maximizes QoS through monitoring various QoS parameters. Also we will briefly explain its difference with that of relational streams.

Section 2 provides an overview for our XML query processing model and the basic differences between XML and relational stream characteristics. Section 3 discusses various QoS parameters that can be measured for XML stream processing systems. Section 4 gives an in-depth view of our proposed load shedding techniques. In Section 5, we provide experimental results that show the effectiveness of our proposed load shedding schemes. Finally we conclude in Section 6.

2. THE XML QUERY PROCESSING MODEL

We use the pipelined query processing model for streamed XML data similar to [4] that requires a smaller memory footprint than the DOM-base query processing. It parses and processes the events of the arriving XML data stream as they become available, only buffering events when necessary. Unlike relational stream processing model, we compile the single-standing continuous query into a network of pipelined operators, without any queues between operators. It is quite similar to the pull-based pipelined query processing model for XML [4]. Each operator acts on the XML events through event handlers. The system compiles a simple XQuery query into an operator pipeline.

The general layout of our XML stream processor is shown in Fig 1. The query processing framework is able to run multiple standing XQuery queries on multiple data stream sources. Our QoS subsystem (QoS Monitor) measures the necessary QoS parameters on the fly and updates and stores them, which in turn are used by the Load Shedding Subsystem (Load Shedder). The QoS monitor continuously monitors input/output rates, rate of the build up of the events in the queue and the rate at which they are consumed by the processing subsystem and it estimates the future load based on various queuing models. There are triggers built into this monitor that control both the Query Scheduler and the Load Shedder. The input streams pass through an intermediate buffer that acts like a platform for load shedding. This platform has a built-in function that monitors the flow of data and decides when to trigger the load shedding process or the query scheduler.

Once the Load Shedder detects congestion, it takes into account various factors like current load, headroom and the necessary QoS parameters of various queries and calculates how much load to shed and in what way. The QoS specifications of the queries determine which methods to follow. We will discuss each of these shedding processes in Section 4 in more detail.

2.1 XML Streams vs. Relational Streams

XML stream systems suffer from similar drawbacks as relational stream systems. But the handling of XML streams is more complex due to (1) Hierarchical and semi-structured in nature, (2) Changing XML element size and depth, (3) Granularity of XML streams. The difficulty in estimating QoS parameters for XML stream systems adds another dimension to this problem.

There is no defined boundary for data elements in XML streams besides the tag structure. However the nested characteristics and semantic dependency of element formations make it difficult to specify a fixed tuple boundary. One solution to this is to convert the XML stream into relational tuples before processing them and to do load shedding if needed by applying relational load shedding techniques. However, this method adds an extra layer of CPU time and increases the latency.

Synopsis construction is a key requirement in any stream processing system that aids in query optimization, load shedding and getting QoS parameter like accuracy. However the synopsis for XML stream is more complicated than their relational counterpart as constructing structural synopsis is an additional task besides the value synopsis [10].

3. QOS PARAMETERS FOR XML
STREAM PROCESSING & QOS MONITORING

Due to the real time nature of the continuous query processing systems, the prevention of any violation of QoS parameters is critical in data stream management systems. The discovery of such violations necessitates continuous monitoring of these parameters and of finding a suitable way to deliver these to the query output with minimal overhead. The most common QoS parameters in the context of XML stream processing are (1) Most tolerable tuple latency, (2) Precision or accuracy, and (3) Maximum memory footprint. While the first two requirements are user-specific, the third one is system-specific and is not controlled by the user. Once the system accepts a query, it becomes a binding obligation on the part of the system to guarantee the required QoS. These primary independent QoS criteria depend on a set of metrics. For example, the tuple latency and memory requirement depend on the characteristics of the processor, the shape, size and complexity of XML input stream, the size and structure of the query pipeline, etc.
The QoS monitoring method we propose is real time with the least possible overhead. Similar to Aurora [1], we measure most tolerable tuple latency and precision in light of a value-based and/or a loss tolerance graph, respectively. The derivation of this value-based and loss-tolerance graph is more complex in an XML stream system because of varying, semi-structured, and often nested elements or tuples. In our framework, we derive the loss tolerance graph or the value-based graph by calculating the utility of XML elements from the output space by correlating them to the query result and by associating a weight or importance to each of the elements in the input space. The more is the weight of an element, the more will be the relative error in the final result if that element is dropped in the process of load shedding. Thus, the objective of this load shedding mechanism is to minimize this weighted relative error by dropping more elements with low weight than elements with high weight. As it will be explained in Section 4, our framework uses a Greedy algorithm for load shedding to materialize the load drop and system recovery. During this phase, it uses the fractional knapsack algorithm to drop the least weighted elements first before attempting to drop the elements with higher weight.

The QoS monitor, calculates the weight for every element. Once a new element comes in through the input stream, it increments its frequency in the input histogram (Histogram_input) and updates value histograms for input for leaf nodes. Similarly, when an element is streamed out as the part of the output, the output structural histogram (Histogram_output) and its value histograms get updated for that elements frequency and value respectively. Once histograms are updated, the QoS monitor updates the relative importance array by recalculating the value for that element and rearranging the sorted array.

As our system does not have intermediate buffers in the operator network, the rate of flow of input should match the rate of event consumption to keep the level of events in the central buffer within an acceptable limit of build up. Besides above parameters, the system also monitors the rate of input, rate of build-up of the buffer level and rate at which the events are being consumed from this buffer.

4. LOAD SCHEDDING TECHNIQUES IN XML STREAM PROCESSING

The purpose of the load shedding is to prevent any congestion situation that might lead to total collapse of the system, rather than to wait for the situation to arise first and take action later. It is always preferable to have preventive mechanism. In general, load shedding implementations can be categorized in two major groups: Syntactic and Semantic. We have implemented a syntactic load shedding techniques in our XML stream system: 'Simple Random Load Shedding' and a semantic technique named as 'Structured Predicate Load Shedding'. The syntactic one is preventive and proactive in nature, the semantic one is reactive.

The first one, which is simple in implementation, arbitrarily sheds the load to prevent a congestion situation. The second one takes into account the structural synopsis and value synopsis to decide which data to shed. Simple Random Load shedding do not take into account the relevance of any dropped data to the result set, the results are approximate with a higher error bound. Relatively the Structured Predicate Load Shedding takes both structural and value synopsis of input and output into account and sheds most irrelevant data first resulting in lower error bounds.

The Structured Predicate Load Shedding differs from the syntactic ones by dropping only irrelevant data and thereby producing more qualitative results. It takes into account complex workload information, structural and value synopsis to decide how much and which ones to shed.

4.1 Simple Random Load Shedder

As shown in figure 4.1, the random shedder functionality is built into the intermediate buffer that collects XML event streams from various sources before passing them to the query processor subsystem. Based on the QoS specifications and current load of the system, if it is decided by the load shedder to go forward with

![FIG 4.1: Random Shedder Implementation in Intermediate Buffer](image)

Random load shedding, it sends a request to the intermediate buffer to go forward to shed some load. Then the buffer drops the first available complete elements, until the load of the system returns back to normal.

Due to hierarchical structure and irregular-grained nature of the XML data streams, it is challenging to implement this element drop. As in the worst case scenario, the stream may be reporting at its deepest level when the trigger comes into action. So the shedder may have to wait for the start tag of a given nesting depth in the stream, in the worst case. Because of this it is preferable to invoke random shedding when the load goes above the threshold.

The XML stream threshold is calculated as \( H^C - D \), where \( H \) is the headroom factor, which is the resource required for steady state, \( C \) is system processing capacity, and \( D \) is the depth of the XML data stream.

Since this shedder drops the elements irrespective of their relevance to the result, it leads to approximate result with high error probability. The intermediate buffer is chosen as the location to drop rather than the operator network as the effect of dropping data at source is more cost effective than dropping them later [1]. Also, by dropping here instead of query pipeline network, we are setting the selection factor to zero for all operators for this entire dropped load. This leads to cleaner implementation that is less invasive and can be managed better at one point.

4.2 Structured Predicate Load Shedder

The Structured Load Shedder is an alternative way to implement Shedding by treating the entire query processing system as a black box as shown in Fig 4.2 and by implementing the shedding system as an overlay by monitoring what is entering and what is leaving the system. The main idea is to maintain an efficient summary or synopsis of the input and output and decide the dropping of
elements based on this summary so that it has the least impact on the quality of the query result.

As explained in [10], structural and value synopsis can be constructed of any XML data. As explained earlier, there are two main parts of this framework. The first is the construction of the appropriate structural and value summary, and the second is the algorithm that performs the shedding based on this summary. There are various methods to maintain a structure-only or a value-structure summary [3, 6, 7, 8, 9, 10, 11, and 12]. We chose to maintain a histogram similar to [5] that has structure only information for all element nodes and separate histograms for values in leaf nodes.

We monitor the input and output streams and their rates closely. Our system builds two histograms or structural synopsis: one for the input stream and another for output stream efficiently. Each histogram partition corresponds to a structural summary node. These histograms map each node from the input/output structural summary into the frequencies of the XML elements that are coming in/out from the input/output stream and are associated with this node. The structural summaries are constructed on-the-fly from the streamed XML events.

The system also builds separate histograms one for each leaf node using the leaf value (Fig 4.3). As an example, a sample XML data tree, shown in Fig 4.3, contains bibliographic data. The structural information of the tree is modeled by assigning unique identifiers or object identifiers to various nodes, such as document-1, author-2, paper-3, name-4, book-5 etc. Similarly the value of ‘title’, ‘year’, ‘name’ etc. are hashed and mapped onto separate value histograms. Similarly, separate structural and values histograms maintained for the output by monitoring the query output.

We prepare a sorted list of elements by their relative importance using both input and output histograms. The relative importance (RI) is calculated by the ratio of its output frequency to its input frequency. RI is zero for an element not appearing in the output result. Similarly, an element appearing in the output but occurs infrequently in the input stream has higher RI. We sort this list of relative importance in ascending order excluding the root element which always has zero RI as it will never be part of the result stream. This sorted list is updated dynamically whenever a new element comes in the input stream or a new element gets out in the result (Fig 4.4).

We utilize this sorted array to make decisions about selecting which elements to drop. The information derived from value histograms best utilized for queries with predicates. We have adopted the greedy method to solve our load shedding problem in line with the classical Fractional Knapsack Problem, as this problem has an optimal greedy solution. The main idea is to shed an item with the maximum value per unit weight (i.e., \( \frac{vi}{wi} \) or value per unit weight). If there is still room available in the knapsack, then the item with the next largest value per unit weight is shed, etc. This procedure continues until the knapsack is full (shedding requirement is achieved).

**Theorem 1.** The Greedy algorithm that selects to remove the element with smallest relative importance results in an optimal solution to XML Stream Overloading problem.

**Proof:** Let there be \( n \) elements in the XML stream besides the root element and they are ordered by their relative importance (RI).

\[
\frac{f_0}{f_1} \leq \frac{f_0}{f_2} \leq \ldots \leq \frac{f_0}{f_n}, \quad \text{where } f_0 \text{ and } f_i \text{ are output and input frequency of element } i \text{ respectively. Each element } i (1 \leq i \leq n) \text{ can have drop fraction of } x_i (0 \leq x_i \leq 1) \text{ such that the total drop will be } X = (x_1, \ldots, x_n).
\]

Let \( X = (x_1, \ldots, x_n) \) be the greedy algorithm solution. For the system that does not have any load shedding due to the existing load is below the system capacity, \( x_i = 0 \) for all \( i \), then the solution is optimal. Also as the system is able to process some fraction of the input, there is no possibility that \( x_i = 1 \) for all \( i \). Thus, some entries of \( X \) can be 1, some other entries can be 0, and there may be one entry with \( x_i < 1 \). Let \( j \) be the smallest value for which \( x_j < 1 \).

![Figure 4.2: Structured Predicate Load Shedder Implementation through Histogram](image)

![Figure 4.3: Unique Identifiers for a sample XML Tree](image)

![Figure 4.4: RI Construction](image)
1. According to the greedy algorithm, if \( i < j \), then \( x_i = 1 \) and if \( i > j \), then \( x_i = 0 \).

Let \( Y = (y_1, \ldots, y_n) \) be any feasible solution. Then we need to show that the quality loss due to \( Y \) is always greater than quality loss due to \( X \).

\[
V(Y) - V(X) \geq 0
\]

The total quality loss for \( X \), \( V(X) = \sum_{i=1}^{n} x_i (f_0/f_i) \)

The total quality loss for \( Y \), \( V(Y) = \sum_{i=1}^{n} y_i (f_0/f_i) \)

\[
V(Y) - V(X) = \sum_{i=1}^{n} y_i (f_0/f_i) - \sum_{i=1}^{n} x_i (f_0/f_i) \quad \text{(1)}
\]

For \( i < j, \ x_i = 1, y_i - x_i \leq 0 \) and \( f_0/f_i \leq f_0/f_j \)

for \( i > j, \ x_i = 0 \) and \( y_i - x_i \geq 0 \) and \( f_0/f_i \geq f_0/f_j \)

Replacing this in equation (1)

\[
V(Y) - V(X) = \sum_{i=1}^{n} y_i (f_0/f_i) - \sum_{i=1}^{n} x_i (f_0/f_i) \geq 0
\]

Hence any solution \( Y \) will cause as much loss as solution \( X \).

The amount of load to shed is the knapsack capacity and selections of elements with lowest relative importance are the ones to be shedded first. If the amount to shed is not met, then the next element in the line, i.e. the element having the next higher relative importance is selected to be shed. The process repeats till the amount to shed is met.

5. PERFORMANCE STUDY

We have implemented both syntactic (Simple Random) and semantic (Structured Predicate) load shedding techniques in Java as separate modules on the top of the core query processor. We have measured the QoS parameters of a workload consisting of a mixture of XPath and XQuery queries running over a representative XML stream data derived from XMark Standard data of 100MB and evaluated them in light of appropriateness of our proposed load shedding techniques. The workload was mixed in order to neutralize the effect of query complexity on the load shedding techniques. A set of 10 queries from each type have been run for each amount of load shedded. The values shown in Fig 5.1 are average of accuracy in terms of element count in the result set for all query results at each load shedding point.

The QoS parameters measured are the accuracy and latency as described in Section 3. We quantified the accuracy in terms of the utility loss for different load shedding methods as well as for a perfect system for comparison. As shown in Fig 5.1, it is clearly evident that structured predicate load shedding is more effective in preserving the accuracy compared to simple random load shedding but with a small overhead of maintenance of related data structures. This overhead can be minimized by efficient implementation. As expected, the loss in accuracy is more prominent at higher loads.

We measured the effects of these two proposed load shedding techniques on the latency of query result by measuring the mean response time of the query processing in milliseconds. As shown in Fig 5.2, the mean response time is always less for structured load shedding relative to the simple load shedding, except at the 35% load shedding point. We are planning to measure these parameters for higher load levels and different query types (join queries) in future.

6. CONCLUSION AND FUTURE WORK

The relational and XML stream processing systems face similar problems but differ in the degree of severity and methods of resolution. As the general relational load shedding frameworks are not valid for XML streams, we presented a general load shedding strategy for XML streams. In addition, we touched upon the list of desirable QoS parameters that need to be measured for XML streams. Finally, we linked these parameters to our general XML load shedding strategy.

Based on our experience, we prove that the semantic framework produces better result than syntactic framework. The simple load shedding, as is used in relational streams [1], leads to loss of structural integrity and violation of the schema.

We are in the process of extending this semantic load shedding framework to cover aggregation and join queries. These frameworks will keep real-time updated structure and value synopsis of all incoming as well as out going elements. Per stream histograms similar to [17] will be constructed and kept for various element values. The victim selection is based on real-time relative weight calculation of various elements taking help of synopsis and choice of an array of novel cost functions. The overhead is optimized to be minimum using intelligent data structures and element selection.

![Accuracy for Simple/Structured Predicate Load Shedding](chart.png)

**FIG 5.1. Effect of Simple Random Load Shedding and Structured Predicate Load shedding on accuracy**
FIG 5.2: Effect of Simple Random Load Shedding and Structured Predicate Load shedding on Latency

7. REFERENCES
15. Mingzhu Wei, Elke A. Rundensteiner and Murali Mani, Utility-Driven Load Shedding in XML Streams, WWW*08, April 2008, Beijing, China.