DELIVERING QOS IN XML DATA STREAM PROCESSING USING LOAD SHEDDING

Ranjan Dash and Leonidas Fegaras

University of Texas at Arlington
Arlington, TX 76019, USA
ranjan.dash@mavs.uta.edu
fegaras@cse.uta.edu

Abstract : *In recent years, we have witnessed the emergence of new types of systems that deal with large volumes of streaming data. Examples include financial data analysis on feeds of stock tickers, sensor-based environmental monitoring, network track monitoring and click stream analysis to push customized advertisements or intrusion detection. Traditional database management systems (DBMS), which are very good at managing large volumes of stored data, are not suitable for this, as streaming needs low-latency processing on live data from push-based sources. Data Stream Management Systems (DSMS) are fast emerging to address this new type of data, but faces challenging issues, such as unpredictable data arrival rate. On bursty mode, processing need surpasses available system capacity affecting the Quality of Service (QoS) adversely. The system overloading is even more acute in XML data streams compared to relational streams due to its extra resource requirements for data preparation and result construction. The main focus of this paper is to find out suitable ways to process this high volume of data streams dealing with the spikes in data arrival gracefully, under limited or fixed system resources in the XML stream context.*

Keywords: *Stream aggregation, multi-dimension, stream cube, group-by query*

1. INTRODUCTION

The data stream is unbounded in nature. Due to very high volume and rate of data generation, traditional data analysis model of DBMS is no longer applicable to this form of data. Unlike DBMS, the data does not need to be stored for future analysis. Streaming data shows temporal essence and needs real-time processing. As a result, DBMS functions such as storing, indexing etc. do not apply to stream data model. Continuously disseminating data sources like sensors need continuous processing. The principle of DBMS query processing does not directly apply to this scenario, requiring a new system: Data Stream Management Systems (DSMS). The DSMS differs from the DBMS in following three ways: First, the volume of the data is so high that the cost for storing (even temporarily) would be out of scale. Second, the data set may be infinite and thus complete storing is out of scope. Third, query processing is expected in real-time or near real-time. The results are supposed to be available as soon as the first data item arrives, thus, the data is said to be transient, whereas the queries running on this data are treated as permanent. More generally, while a DBMS works in a query-driven fashion, the DSMS puts the focus on data-driven query evaluation. Unlike pull-based DBMS, the DSMS is push based and is vulnerable to data flow characteristics of input.

The data stream applications are ubiquitous and found in area of Fraud detection applications (to monitor ATM and Credit Card transactions for abnormal usage behavior), Click Streams (to push customized advertisements in web pages), Online auction applications (like eBay), Call Detail Record [19] (to generate billing information real-time), weather data sensors (Weather Monitors, Habitat monitoring, biomedical monitoring, road traffic monitoring, RFID-based asset tracking, GPS-based location tracking, network traffic monitoring etc.), stock trading, News alerts (RSS feeds), Event composition in CEP (Complex Event Processing) .

Use of various statistical data summaries known as synopses is very extensive in the field of data streams, data mining and DBMS. Some key synopsis methods such as sampling, sketches, wavelets and histograms have been successfully used in data stream area. They are used for approximate query estimation, approximate join size estimation, computing aggregate statistics, such as frequency counts, quantiles and heavy hitters, change detection. The design characteristic of any of these data summaries are time and space efficient, evolution sensitive and satisfy the one pass constraint of data streams. While methods such as wavelets and sketches are complex, sampling and histograms are simple and straight forward. While sampling preserves the multidimensional representation of the original data stream, histograms abstracts the behavior of more complex synopsis types such as wavelets (Haar wavelet coefficients) and sketches. In simple term, a histogram is the representation of data distribution along any attribute by means of dividing the attribute range into a set of ranges (buckets), and maintain the count for each bucket. The source of inaccuracy in the use of histograms is that the distribution of the data points within a bucket is not maintained [5]. So bucket design is an important consideration in histograms. Both structural and value synopsis of a XML data can be constructed and maintained using histograms [55, 56].

1.1 Motivation

Some of the distinct characteristics of data streams are, continuous processing tuples, continuous or long standing queries, tolerance to approximate results, real-time or near real-time processing (minimal tuple latency), and dependency on data input or arrival rate, delivery of QoS metrics to query processing. The common QoS parameters are Response Time or known as tuple latency, Precision or accuracy of the result, Throughput or number of output tuples per unit time, Memory Usage, and output flow characteristics.
To overcome the overload challenge, a DSMS continuously monitors the system capacity and carry out run time optimization of queries. With a fixed system capacity, adapting it to the increased system load is not an option but dropping excess load (load shedding) is a viable solution. Dropping input data to reduce tuple latency without sacrificing the result accuracy is the biggest challenge in data stream systems. The established solution to load shedding is to introduce load shedders (sometimes known as drop operators) into the query plan. But these drop operators add an extra overhead in planning, scheduling and activating or deactivating them to already resource starved system.

Unlike relational streams, tuples (elements) in XML stream are structurally variable in size and content. The processing of XML stream tuples has additional complexities that add to the processing overhead. Conversion of these elements to equivalent relational tuples and process them using relational techniques adds an extra overhead [73]. Alternately, the element construction from XML stream events such as SAX events to regular XML elements also adds an overhead. Thus, in addition to overall streaming challenge, XML stream systems have to process queries with different semantic complexity using nested hierarchical tuples while delivering guaranteed QoS.

1.2 Our Approach

We present a framework for XML data stream processing in this paper. We cover different aspects, such as query processing (XQuery processing of XML data streams using pipeline methodology [31]), QoS delivery through different load shedding schemes. We have implemented load shedding methods for various query types in XML data streams. We use simple structural and value summaries to help devise these load shedding strategies [25-27]. We compare our stream synopsis construction and maintenance to other existing systems and prove its effectiveness from space and time complexity standpoint. We use simple histogram techniques to create structural and value synopsis of both input and output data streams and use them to decide victim selection. Though we use XML streams to prove our framework's effectiveness, yet it can be applied to relational streams due to its general nature.

The rest of this paper is organized as follows. We cover the related works in the field of QoS in data stream processing and XML streams in Section 2. The Section 3 covers the architectural lay out of our system. We describe the load shedding in set-valued queries in Section 4. The solution frameworks for join and aggregation queries are covered in Sections 5 and 6 respectively. The experimental results for all these types of queries are covered in Section 7. We conclude in Section 8 with future work propositions.

2. RELATED WORK

Quality of service (QoS) was identified as an important attribute of overall performance measure in data stream systems by [17], as it is implemented as an integral part of Aurora system through QoS monitor [17, 14, 3, 72, 65]. Aurora defines QoS in terms of response time, tuple drops and output value produced, through a set of two dimensional graphs, comprising of Delay-based (response time), Drop-based (tuple drops) and Value-based (output quality) graphs. They termed these graphs collectively as QoS graphs or QoS data structure. Delay-based graph is used to determine when to initiate the load shedding while drop-based and value-based graphs are used to manage the load shedding once it starts.

Stanford Stream Data Manager (STREAM) [10, 2, 13, 9, 11, 8, 53, 12] also addresses issue of resource management such as maximum run-time memory etc. in the context of constraints such as maximum latency [9].

They emphasized the importance of quality of service in [10]. Their sampling based load shedding to deal with resource issues is covered in [11]. MavStream [39, 41] is designed as a QoS aware data stream management system. It addresses the issue of QoS through capacity planning and QoS monitoring in a more systematic manner. The system uses a whole set of scheduling strategies to deliver best QoS to continuous queries. A separate QoS monitor interacts closely with run-time optimizer for various QoS delivery mechanisms. They have designed their QoS mechanism using queuing theory [40] and addressed the load shedding mechanism extensively [42, 46].

Different forms of XML data, fragments, SAX events and their handling is covered in [15, 16]. In addition to the structure and data content complexities, the XML stream has to support all streaming challenges like a relational stream. So the XML stream system has to support all QoS metrics similar to the relational stream and has to deal with data metrics typical to XML data. Those are: (1) tuple size (number of elements and attributes), (2) tuple structure (number of recursions and REFs and IDREFs), (3) tuple depth (tree depth), (4) fan-in (number of edges coming out of an element), (5) fan-out (number of edges coming into the element). The quality of service is well researched in XML field for various dissemination services such as pub/sub and selective dissemination of information (SDI) systems [60]. Besides the complexity of data, the query complexity (XPath and XQuery constructs) also plays a key role in affecting the overall QoS of the system. Authors in [69] consider query preferences (quantitative and qualitative) similar to QoS specifications to ensure best qualitative result to queries in XML stream.

Synopsis structures, such as sampling, wavelets, sketches and histograms are more popular in data streams compared to the DBMS. The technique of synopsis has been successfully used in the area of query estimation [18], approximate join estimation [7, 29, 30], aggregation statistics such as frequency counts, quantiles and heavy hitters [20, 22, 51, 50], stream mining methods such as change detection etc.[4, 61]. Histograms have been widely used in data stream systems in many ways such as equi-width, equi-depth and V-Optimal histograms. Guha et al. have proposed construction of V-Optimal histograms ($(1+\varepsilon)$-optimal histogram) over data streams [35]. Construction of wavelet based histograms on data streams dynamically is proposed in [6]. Both structural and value synopsis of a XML data can be constructed and maintained using histograms [55, 56]. Though, the synopsis structures have been used successfully in various fields of data stream systems, their application in delivering quality of service in data stream systems is non-existent. To the best of our knowledge, our load shedding framework is the first one that takes the help of an intelligent synopsis structure to effect a load shedding strategy [25-27].

Load shedding has been implemented as part of delivering QoS to standing queries in Aurora [65] and STREAM [11]. Load shedding mechanisms in both is based on near past statistics. STREAM system formulates the load shedding problem as an optimization problem, which calculates the optimum sampling rate (by which a tuple is discarded) at a load shedder so as to minimize the overall relative error. Aurora [65] differentiates its load shedding mechanism from STREAM in the way that it is more quality of service or utility to user based, where it allows users to specify them. The overall scheme is a greedy algorithm that strives to maximize the amount of load reduced while minimizing loss of utility (QoS) [65]. Load shedding in MavStream [42, 46] is formulated around general continuous queries in conjunction with scheduling strategies so that violation of predefined QoS requirements are prevented by discarding some unprocessed tuples. Wei et al. in their papers [68, 69] have proposed a framework that enacts load shedding through user based preference model. It collects user preferences to rewrite XQueries into shed queries and does load shedding in automaton. The issues with this type of strategies are extra computation of query rewriting and partial processing of tuples before dropping. A framework that works on both load shedding and load spilling to reduce the burden on resources during bursty overload has been proposed for XML streams in [70].

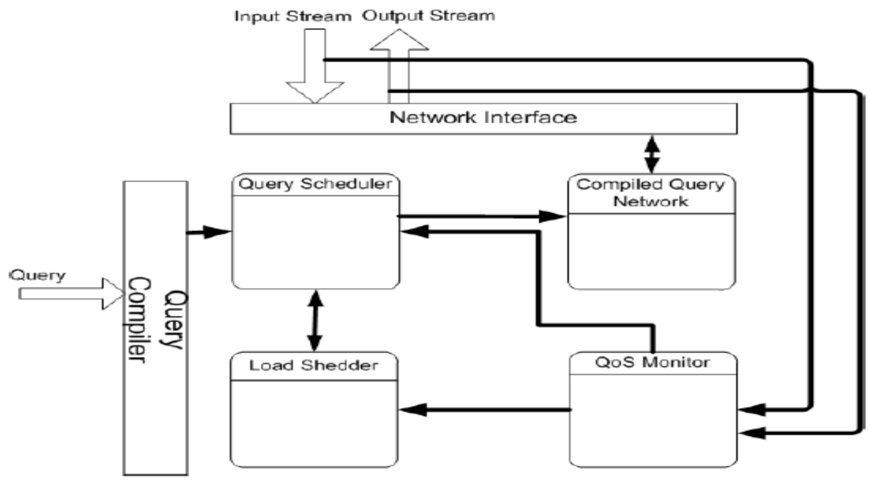


Fig. 1. Overall XML Stream Processing Architecture

Data Stream Joins Joining potentially require unbounded intermediate storage to accommodate the delay between data arrival. Thus the use of various approximation techniques, such as synopses, are common in data stream joining that produce approximate answers. Ding et al. [28] have prescribed punctuation-exploiting Window Joins (PWJoin) to produce qualitative result with limited memory space. An age-based model has been introduced in [63] by approximating a sliding-window join over a data stream. Joining streams under resource constraints has been addressed for relational streams [32, 47, 63, 24] using windowed joins. Solution through Load shedding in relational stream joins has been addressed by [32, 63] whereas through randomized sketches in [47]. Kang et al. study the load shedding of binary sliding window joins with an objective of maximizing the number of output tuples [45]. Multi-join algorithms with join order heuristics has been proposed by [34] based on processing cost per unit time. The concept of load shedding in case of multiple pair wise joins is discussed in [47]. The problem of memory-limited approximation of sliding-window joins for relational streams has been discussed in [63]. Joining XML streams for a pub/sub system is proposed through Massively Multi-Query Join Processing technique [38].

Data Stream Aggregations Aggregation queries including group-bys are more complex than other stream operations, because of their blocking nature and requirement of unbounded memory. They have been classified into various aggregation function bases: distributive, algebraic, and holistic [8] and solved using concept of sliding windows. Processing complex aggregate SQL queries over continuous data streams with limited memory is presented in [29]. It uses randomized techniques, such as sketch summaries, and statistical information in the form of synopsis, such as histograms, to get approximate answers to aggregate queries with provable guarantees using domain partitioning methods to further boost the accuracy of the final estimates. A load shedding method that uses a drop operator for aggregation queries over data streams is described in [64]. It uses drop operators called Window Drop that is aware of the window properties to effect its load shedding. In Aggregation Join queries, relative location of aggregation vs. join in query plan, is an interesting problem. Many aspects of aggregation Join queries have been covered in [21, 43, 66]. Query optimization and query transformation of continuous aggregation join queries on data streams is covered in [66].

3. SYSTEM OVERVIEW

Our implementation is an overlay on the core query processing engine. We use the pipelined query processing model for streamed XML data that requires a smaller memory footprint than the DOM-based query processing. It parses and processes the events of the arriving XML data stream as they become available, only buffering events when necessary. We compile the single-standing continuous query into a network of pipelined operators, without any queues between operators. Implemented on the basis of pull-based pipelined query processing model, each operator acts on the XML events through event handlers. The system compiles a simple XQuery query into an operator pipeline.

Figure 1 illustrates the overall architectural layout of our stream processing system. The query processing framework is able to run multiple standing XQuery queries on multiple data stream sources. Similar to any stream processors, our system has basic query processing components such as Query Compiler, Query Scheduler and Query Network. The main components that directly influenced by our framework are QoS Monitor and Load Shedder. The QoS Monitor measures the necessary QoS parameters dynamically against the QoS requirements of the system. The QoS Monitor continuously monitors input/output rates, rate of the buildup of the events in the queue and the rate at which they are consumed by the processing subsystem and estimates the future load based on various queuing models. The Load Shedder uses feedback of QoS values from QoS Monitor to effect load shedding. 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.

3.1 Data Model

XML data often modeled as node labeled tree such as Document Object Model (DOM). But as streams are unbounded, in our system, we model the XML stream as an infinite sequence of events similar to SAX events [52]. Each event is modeled as a quadruple of the form (*i; t; l; al*). (1) The *i* denotes the stream identifier. It helps uniquely identify the source stream in case of multiple stream sources. (2) t is the tag or the name of the element that generates the SAX event. (3) l is the level or depth of the element in the document tree. (4) The al being attribute list for the element, is modeled as an array of (n; v) pairs; where n indicates the name of attribute and v its corresponding value. Elements that have no attributes will have a null list.

3.2 Challenges of XML Streams

The major challenge in XML stream processing system is the irregular size and form of its tuple. Unlike relational streams where tuples are well-defined, XML elements in XML streams are irregular in size. There is no defined boundary for data elements in XML streams. The nested characteristics and semantic dependency of XML elements make it difficult to specify a fixed tuple boundary. As we convert SAX events into elements, we give each element an unique id that helps us in making the structural summary of the stream which plays a critical role in our framework. Besides the streaming challenges of relational streams, the XML stream have to face other challenges, such as (1) Hierarchical and semi-structured form of data, (2) Changing XML element size and depth, (3) Nested and non-uniform Granularity of XML streams.

3.3 Synopses for XML streams

Various types of synopsis (sampling, wavelets, histograms and sketches) have been used for various problems in stream systems. The synopsis construction has to satisfy certain resource critical conditions such as bounded storage, minimal per record processing time and single pass criteria. Of these, 1-d histograms are the most basic types of synopsis that approximate the frequency distribution of an attribute value. Though these types cannot capture the inter-attribute correlations, but given super linear space complexity, they capture the frequency distribution of any attribute most effectively. Though equi-depth and V-Optimal histograms preserve the distribution of data items within a bucket, they are computation intensive, thus are excluded from our preferred set of tools. We have used the equi-width histograms in all our frameworks. This not only gives us a clear summary of data distributions from a bigger picture is concerned, but also very simple to construct and maintain and consume least resource overhead. We are aware of the fact that equi-width histograms are not good for range queries compared to their counterparts, such as equi-depth or V-Optimal ones. But we limit the errors in intra-bucket distribution through suitable bucket size selection. Also the choice of equi-width histograms has little or no effect in case aggregation queries and join queries.

Structural Synopsis Given the complexity of XML data, the synopsis for XML stream is more complicated than their relational counterpart. To capture the entire essence of the data both value and contextual, we construct the separate but inter-related synopses; that we call structural and value synopsis. The structural synopsis records the frequency of all unique paths in the XML tree or a frequency distribution of the structural summary. The structural summary and its corresponding structural synopsis for the sample DBLP XML benchmark data are shown in Figures 2 and 3 respectively. Each node or element in the tree carries its corresponding tagname and an unique id.

Value Synopsis Similar to synopsis for an attribute value in relational stream, the value synopsis for any text node in the XML tree can be constructed. We construct the separate value synopsis for each unique path that contains a text node. The corresponding value synopsis for the DBLP XML benchmark data is shown in Figure 4.

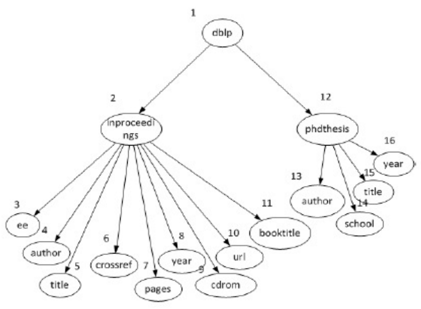


Fig. 2. Structural Summary of an XML document

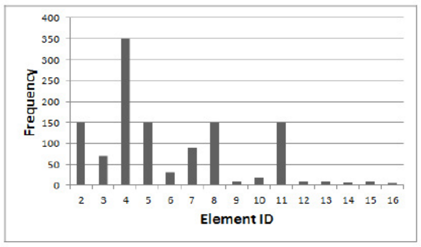


Fig. 3. Sample Structural Synopsis of an XML document

4. LOAD SHEDDING IN XML STREAMS FOR SET-VALUED QUERIES

Besides the QoS parameters, we consider the utility metric as one important QoS parameter for XML streams. As XML streams are more complex and nested as far as the tuples are concerned, the utility of any tuple not only depends on the textual value of that element, but also on its structural position with respect to the overall tree. Thus we have introduced a new QoS metric for XML streams called Utility metric. The utility of an XML element is calculated from its presence in output space and 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.

The Utility Metric is captured through a set of intelligent synopses which in turn implemented through a suite of innovative data structures. We have built both structural and value synopsis for both input streams and output streams. The synopses set for our stream processing system are divided into two sets of 1-d histograms; one for the input and the other for output streams. In each set, we capture the structural synopsis of the XML stream and the value synopses for each of its text nodes as shown in Figures 3 and 4 respectively.

Our objective is to carry out the load shedding with least possible overhead. QoS specifications are user transparent and system configuration-based in our system. 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 structural synopsis ($\text{Histogram}_{\text{input}}$) 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 ($\text{Histogram}_{\text{output}}$) and its value histograms get updated for that elements frequency and value respectively.

4.1 Load Shedding in XML Stream Processing

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 shedding framework 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. Our suite of synopses are geared towards maintaining and updating the relative weights or utilities of all unique elements in XML tree. When an element gets shedded as part of the load shedding, there are some error in the result set as this element which would have contributed towards the result got shedded. Our main objective is to minimize this weighted relative error by dropping more elements with low weight than elements with high weight.

We have implemented two different types of load shedding mechanisms as covered in following subsections. They can be categorized into two major groups: Syntactic and Semantic. We have termed the syntactic load shedding technique as Simple Random Load Shedding and the semantic technique 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. Our load shedding system gets triggered by the QoS monitor that detects the congestion. The load shedder takes into account various factors like current load, headroom and the necessary QoS parameters of the system and calculates how much load to shed to keep the system functioning in an acceptable manner. Based on the criticality of the congestion or configuration, the system enacts which way to do the load shedding. Our system takes a holistic approach and formalizes the load shedding as an optimization problem, deciding which tuple to shed. This solution is based on admission control paradigm and tries to save more resources restricting suitable tuples ahead of their admission into the system. We treat the whole processing system as a black box and the shedding as a layer on the top of it. We have a buffer external to the operator network that buffers the SAX events and creates tuples from them. Our load shedding mechanism is built as part of this buffer called intermediate buffer. The rate of flow of input should match the rate of event consumption to keep the level of events in this intermediate buffer within an acceptable limit of buildup. 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.

Simple Random Load Shedder As shown in Figure 5, the random shedder functionality is built into the intermediate buffer that collects XML SAX 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 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 [65]. 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.

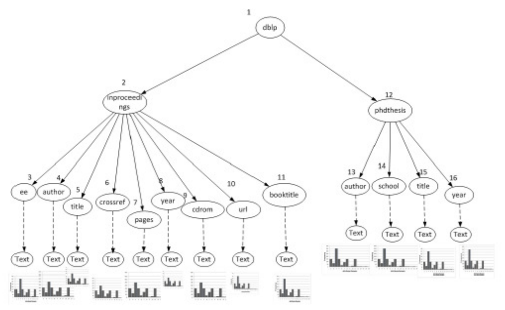


Fig. 4. Sample Value Synopsis of an XML document

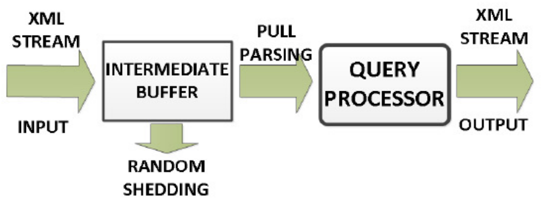


Fig. 5. Random Shedder Implementation in Intermediate Buffer

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 Figure 6 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 earlier, there are two main parts of our 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. We chose to maintain a histogram similar to [49] 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-y from the streamed XML events. The system also builds separate histograms one for each leaf node using the leaf value (Figure 4).

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 (Figure 7).

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., *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).

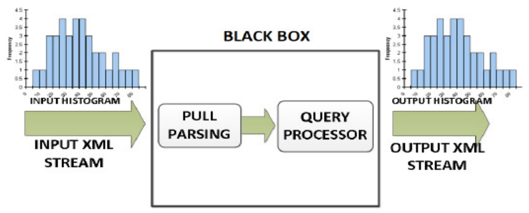


Fig. 6. Structured Predicate Load Shedder Implementation through Histogram

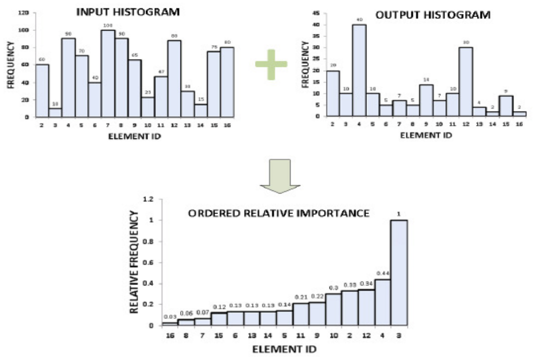


Fig. 7. Relative Importance Construction

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 (R.I).

$f_{o1}/f_{i1} \leq f_{o2}/f_{i2} \leq .. \leq f_{on}/f_{in}$ where f_{oi} and f_{ii} are output and input frequency of element i respectively. Each element i $(1 \leq i \leq n)$ can have drop fraction of x_i $(0 \leq x_i \leq 1)$ such that the total drop will be $X = (x_1, .., x_n)$.

Let $X = (x_1, ..., x_n)$ 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 $x_j < 1$. According to the greedy algorithm, if $i < j$, then $x_i = 1$, and if $i > j$, then $x_i = 0$.

Let $Y = (y_1, ..., y_n)$ 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 $V(X) = \sum_{i=1}^{n} x_i(f_{oi}/f_{ii})$ The total quality loss for Y; $V(Y) = \sum_{i=1}^{n} y_i(f_{oi}/f_{ii})$

$$V(Y) - V(X) = \sum_{i=1}^{n} y_i(f_{oi}/f_{ii}) - \sum_{i=1}^{n} x_i(f_{oi}/f_{ii}) \tag{1}$$

For $i < j, x_i = 1; y_i - x_i \leq 0 and f_{oi}/f_{ii} \leq f_{oj}/f_{ij}$ for $i > j, x_i = 0$ and $y_i - x_i \geq 0$ and $f_{oi}/f_{ii} \geq f_{oj}/f_{ij}$

Replacing this in equation (1)

$V(Y) - V(X) = \sum_{i=1}^{n}(y_i - x_i)(f_{oi}/f_{ii}) \geq \sum_{i=1}^{n}(y_i - x_i)(f_{oj}/f_{ij}) \geq 0$

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. Load Shedding In XML Stream Joins

We propose a unique windowing technique based on an innovative cost functions for join query processing under memory constraints. The logical window construction is controlled through unique data structure and maintained using load shedding technique with least overhead. We show our strategy through the accuracy of the join result from two XML streams using standard XQuery. Our window is built on the concept of relevance rather than time or frequency. We utilize the concept of structural and value synopsis that we covered in the previous section. Our framework is built on the basis of both frequency based and age based models [63]. The load shedding is driven by this cost based element expiration. We have extended the concept of sliding window to a logical window that is fixed in size and sheds tuples based on relevance rather than time or frequency.

5.1 XML Join Query Processing Model

The frequency based model does not work for all kinds of streams [63]. It fails for streams where the relevance distribution is skew over the life of the tuple as in the case of on-line auction scenario where more bids come towards closing of an item. The age based models require more monitoring and frequent adjustment to window mechanism to yield max sub set result [63]. Depending on the age curve shape, different strategies to be followed to optimize the result. On the other hand if time based window model is followed, it will result in decreased productivity due to discard of relevant tuples when the resource is limited. Hence the solution to optimize the productivity in a resource crunched situation is to collect all relevant tuples while discarding the irrelevant ones. The relevance is decided based on its probable participation in the join process.

We have a new window model to achieve the max sub set result or optimize the productivity. The model ensures the high usefulness of all stored tuples while making a judicious decision on load shedding. We have come up with a framework to measure relative relevance of various elements and thereby making the shedding decisions wisely. We have formulated a unique cost function that is central to this model and an innovative data structure that is very efficient in implementing the framework.

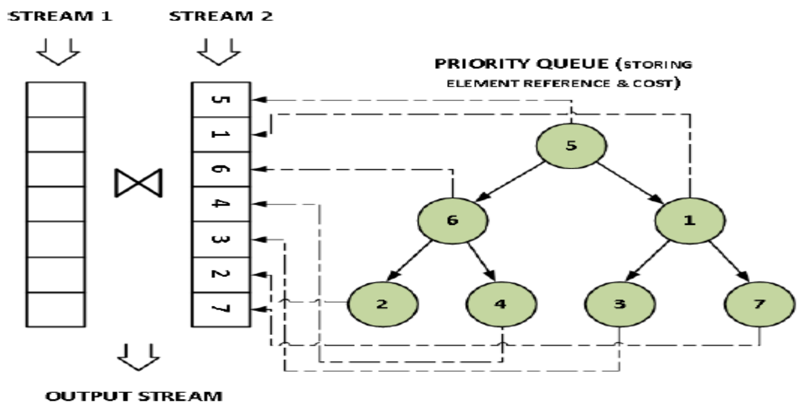


Fig. 8. Overall XML Stream Join Query Processing Architecture

5.2 Synopsis for Join Processing

The synopsis construction is light and has least overhead for its maintenance. The heart of the synopsis is the construction and maintenance of the heap based priority queue with relevance measure of elements of the stream. The relevance measure of each element is calculated using the cost function that is described in the following section. Each time an element comes in any stream the cost of it is calculated and updated or inserted into the heap based on its preexistence in the heap. So also the cost of all nodes gets updated at this instance. The heap is kept updated with the node with least value at the root, ready to be sheded if need arises.

Cost Function The calculation of weight for each element in the logical queue is done using the following cost function [Equation 5.1]. This has two significant parts: (a) *Age factor*, and (b) *Utility factor*. The age component is derived from forward decay based model [23] which is in turn based on the philosophy that old data is less important. The utility part is derived from the intuition that any tuple or element that takes part successfully in a join is useful. The degree of usefulness or relevance is based on the context of the stream. A stream that has time sensitive data might have lower utility for future joins even if it has taken part in the join at present. So depending on the type of stream these factors contribute differently towards the overall relevance. The cost of relevance for an element ei of stream Si that has arrived at time ti and measured at time t such that t >= t_i and t_i > L is

$$C = C_1 f(t_i, t) + C_2 f(u_i) \tag{2}$$

Where
$f(t_i, t) = g(ti, L)/g(t-L)$ a decay function for an element with a time stamp t and land mark time stamp L
$f(u_i)$ = Count of times that this element has been part of the join
C_1, C_2 are arbitrary constants whose values can be tuned based on data type of XML stream.

Age based Relevance This part influences the overall relevance through age based load shedding model. Based on the stream context we are using the following three decay functions the value of which varies between 0 and 1.

Linear Decay - This follows a linear distribution for the weight with any element that arrives at the time of measuring is having a weight of 1 and that has arrived at the beginning of the system start (landmark time L) with a weight of 0. The function is as follows.

$$f(t_i, t) = \frac{(t_i - L)}{(t - L)} \tag{3}$$

At t = ti, the weight is 1 and as t increases it decreases linearly. Most of the normal streams follow this decay pattern.

Polynomial Decay

$$f(t_i, t) = (t_i - L)^2/(t - L)^2 \tag{4}$$

Exponential Decay - We have found out much of resource (both space and time) can be freed up using the exponential decay with reasonable impact on the recall.

$$f(t_i, t) = \exp(t_i - t) \tag{5}$$

Utility based Relevance The second part of the cost function comes from the utility of each element from their participation in join operation. If the element is a subset of the join result it bumps the count. We have implemented this part as a simple count in our experiment. This part represents a simple form of the output history. More accurate weights can be calculated based on the timestamps of its appearance in the result stream.

Relevance based Window The priority queue that is implemented as a heap acts as a logical window for the input stream. Its size determined by the memory availability decides when to start the shedding. The victim selection is facilitated through the heap structure to shed the lowest weight element at the root. As the relative weight of each element node driven by the choice of cost functions discussed previously, the age of the element and number of times its appearance in the result stream play a crucial role in determining which element to be kept in the window irrespective of their arrival.

5.3 Load Shedding Mechanism for Join

Our basic algorithm for executing join *S1[W1]XS2[W2]* is shown in Table 2. If memory is limited, we need to modify the algorithm in two ways. First, in Line 2, we update $S_1[W_1]$ in addition to $S_2[W_2]$ to free up memory occupied by expired tuples. More importantly, in Line 4, memory may be insufficient to add s to *S1[W1]*. In this case, we need to decide whether s is to be discarded or admitted into *S1[W1]*, and if it is to be admitted, which of the existing tuples is to be discarded. Due to load-shedding, only a fraction of the true result (recall) will actually be produced.

We adopt a window based approach to process the join query. The window in our case is a fixed size buffer for each source stream. The data is kept in the form of a heap data structure of elements sorted by the cost of each element. The least cost element remains at the top ready to be shed. The shedding action is triggered as a complete element arrives at the source. Upon arrival, the cost is updated for the new element

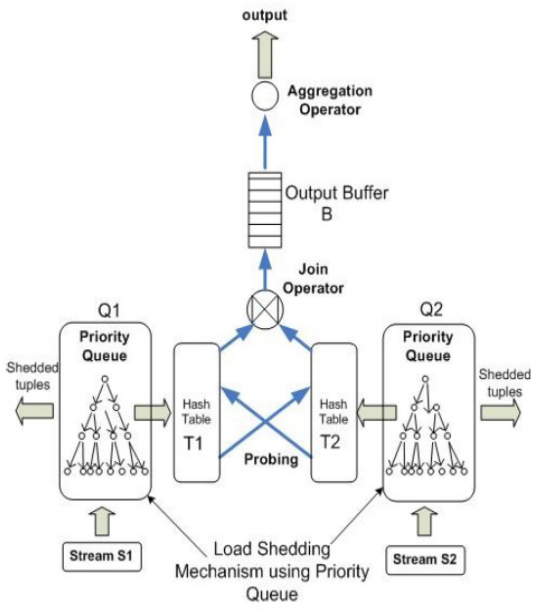
Table 1. Join Execution Steps

```
Ni: Max size of logical buffer Si[Wi], Qi: Associated relevance queue of stream Si
Bi: Associated hash based buffer of stream Si, fi: Hash functions for streams Si. For simplicity we kept
f1 = f2
attri: join attribute of stream Si
1. When a new element e arrives in any stream S1
Phase I: Symmetric Hash Join
2. Calculate hash value of e by applying f1 on attr1 and insert in B1.
3. Calculate hash value of e by applying f2 on attr2
4. Probe B2 using the value from step 3.
5. Emit the result
Phase II: Synopsis Construction
6. If size of Q1 < Ni, Insert into Qi
7. Else a. calculate cost of e Ce
b. If Ce < cost of element at head of Q1, throw e and remove the corresponding element from B1
c. Else shed head of Qi, Insert Ce into Q1
(Similarly repeat the steps for any element that reaches in stream S2 symmetrically with converse data
structures)
```

-if it matches with an existing element in the heap, else the new element is added to the heap. If the action is update, the heap is re-sorted through heapify operation. On insert, the shedding decision is taken if the buffer is already full. If the buffer is not full, the element is added to the heap and heapified. The shedding is decided if the cost of the new element is more than the cost of the element at the top of heap. Else the new element is dropped without any shedding. The shedding enacts the deleting of the element at the top and adding of new element to the heap. Once again the heap gets heapified after the operation completes. For simplicity, we have processed the join between two data stream sources. We implemented our join query processing as a symmetric hash join between our two window buffers. The reference of these elements has been maintained in respective hash tables for faster access.

6. LOAD SHEDDING IN XML STREAM AGGREGATIONS

There are various stream-based commercial applications like eBay that require to identify best seller items on the y and to maintain a list of top-k selling items as sales continue. The determination of top-k items is significantly harder depending upon the rate of auctions opening and closing, variability of biddings and the numbers of auction items. The main difficulty in maintaining these bestseller items is in detecting those infrequent values that become significantly frequent over time. This inherently requires to process continuous aggregation queries with limited memory. The problem becomes even harder when data in another source (stream) decides which item qualifies as more frequent than others. We propose a novel data shedding framework that uses an innovative synopsis to answer such queries under resource limitations. The synopsis takes into account the data distribution and cardinality of attributes and is very light weight to create and maintain. We proved the effectiveness of our strategy using XML stream domain and by measuring the accuracy of the result after processing joins and top-k aggregation queries on these streams.
As the aggregation is a blocking operation, there is a trade-off on size of the window and the time basis for the expected result. The size of windows gets limited by the available resource which affects the quality of the outcome adversely. So there is always a trade-o_ between the quality of the result that you want verses the amount of memory that you can give to such solutions. We propose a solution that is constant in memory and provides acceptable result accuracy using a stream cube synopsis. Our framework does state management between two streams to provide the top-k list dynamically. The synopsis is built over the notion of data distribution of joining attributes. We use the existing XMark auction data [59] to prove the effectiveness of our proposal.



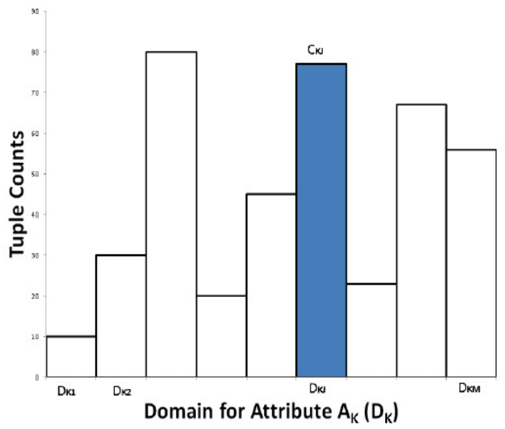


Fig. 9. Overall Aggregation Query Processing Architecture **Fig. 10.** Compressed Domain with discrete sub-domains

6.1 Aggregation Query Processing Model

The sliding window W_i for stream S_i (denoted as $S_i[W_i]$) which may be tuple based or time based, is the basic traditional stream processing unit. But, storing tuples from individual windows into a substantial future, for future joins or group-bys, requires huge memory. For example, in case of auction and bid stream [59], the auctions have to be stored for incoming bids in bid stream, which might come at a later time in the future, well past the present window of auction stream. This necessitates storing items in auction stream beyond their window where they appear to have a possible join in future with bid stream. This either requires an extended window or an indirect solution through punctuated stream for a qualitative result. This poses a challenge to get accurate result with a limited space (memory). Besides, the possible number of group-bys that can be applied to a stream with a *n* number of attributes $\{A_1; A_2; ..; A_n\}$ or dimensions having domain $\{d_1; d_2; ..; d_n\}$, can be 2^n [37].

In naive way, we can aggregate the bid stream on a window that spans from open auction to closed auction of the corresponding item. To determine top-k hot items, these windows must span over the life span of all auction items. However the problem becomes much simpler to answer when asked what are the top-k hottest items in last one minute or so [67]. Even for this, the memory requirement may be exhaustive if number of items n becomes very large. The space complexity would be of size of $\Omega(n)$ [33]. We try to solve this by reducing this number n through reducing the domain size of the auction items but effectively preserving the counts of individual data items. But in our approach, the size of the window Si[Wi] is determined on the basis of the available memory from resource limitation aspect. In case of exact analysis, the window should be enveloping all possible time spans of auction items in discussion. But for case of approximate analysis of top-k items, we determine the size of the window be much less than that should be to save all tuples. We have made the size of window, a function of the `k' based on the top-k value.

6.2 Application of Compressed OLAP Data Cubes as an effective stream synopsis

Compressed data cubes have been used as an efficient tool to answer aggregate queries. Multivariate Gaussian probability density functions have been used to answer aggregate queries over continuous dimension without accessing the real data [62]. This technique is only valid for continuous dimensions like age, salary etc. but dimensions such as item category etc. are discrete and will not follow the pattern. Various compression techniques have been discussed in [71].

We have used a hash function (*mod x*) to compress the cardinality of any attribute e.g. the item types in the auction stream. The value of x can be considered based on the space that can be allocated for the synopsis. We have taken it as 100 for both streams for example in our experiments. Based on this value, the domain is divided into 100 sub-domains. We have adopted the OLAP domain compression method as described in [62] to formulate this compression technique to reduce the space requirement for maintenance of counts of bid. We have used a cost function as shown in Equation 6 which is based on the count to calculate the cost factor of an incoming element. The element with this cost is fed into the priority queue depending on the value of the cost associated. This relative cost of the element helps drive our load shedding process before the element is decided as an useful element and then is inserted into the hash buffer that process the symmetric hash join with the other buffer.

6.3 Synopsis for Aggregation

As our aim is to get the top-k hot items being auctioned, we can get a data distribution, where the items in each category can be assumed to be uniformly distributed. The total number of hits in any category c can be computed as a function of the probability distribution function of hits, P*r(c)* in each category where item a and item b are the beginning and end items in the category respectively as:

$$N. \int_{item_a}^{item_b} Pr(c)\, dc.$$

Based on this value, it is easy to calculate which are top-k selling categories. Similarly it can be done for top-k items in each category. We transformed this continuous model into a discretized model to keep the counts of items (bids), that not only saves us huge space but also helps us in fast calculating the probability of shedding for each tuple. The heart of the synopsis is the construction and maintenance of this histogram like data structure on compressed dimensions.

6.4 Domain Compression

The use of stream cubes in analysis of streams is a topic well covered in [36]. Use of cubes to answer aggregate queries has been discussed in [37]. Various domain compression techniques are followed in the field of OLAP to save space for data cubes [62, 71]. The compression is achieved by exploiting the statistical structure of the data and its probability distribution along the domain. It works well where the density distribution is continuous such as salary, age etc. We have adopted a similar approach, but in our case where the attribute (*itemid*) might not be continuous, we follow the discretized model of domain compression and used histogram model instead. Let the stream S_i has domain D_k for its attribute A_k. We have used a hash function as described in 2.1 to bring in this domain size D_k to manageable range and further divide it into m sub-domains $\{D_{k1};D_{k2}; :::D_{km}\}$.We maintain counters $C = \{C_{k1};C_{k2}; . . .C_{km}\}$ for each sub-domain that gets incremented as bids with corresponding itemid flows in. Based on the value of these counters the probability of shedding is decided. The domain compression not only saves us space for the synopsis construction and maintenance but also provides faster cost evaluation for shedding.

6.5 Cost Function

The calculation of weight for each element in the logical queue (priority queue) is done using the following cost function *(Equation 6.1)*. Keeping top-K nature of query in view, we have decided to shed non-frequent items before they consume critical computational resource. We have calculated probability of shedding for each incoming bid *(Equation 6.1)*. Based on the count of past similar tuples, its cost goes up or down and so also the chance of it getting shedded.

Let an element E_i arrives in stream *Si*. The desired dimension k for S_i whose domain is D_k which in turn is divided into M sub-domains $\{D_{k1};D_{k2}; :::D_{kM}\}$ as shown in Figure 2. Depending on the attribute value for this dimension, Ei hashes to sub-domain D_{kj}(j = 1..M). Let the count of elements already in D_{kj} is C_{kj} . The probability of shedding P_{Ei} of an element Ei is as follows.

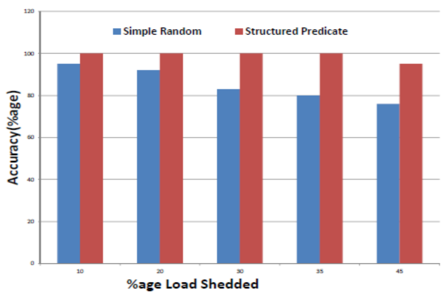
$$|P_{Ei} = 1 - p(\frac{C_{kj}}{\sum_{j=1}^{M} C_{kj}}) \tag{6}$$

Table 2. Execution Steps (Refer to Figure 9 for notations used)

1. A new element e arrives in any stream S2 (the bid stream)
Phase I: Shedding and updating the data structure Phase
2. If the element is already in hash table T2
a. Update the element cost in the priority queue Q2.
Else
b. Calculate the cost (Ce) of the element e, and try insert it into Q2 (as in step 3).
3. If Ce > cost of element at head of Q2,
Throw element e without insertion
Else
Shed head of Q2 and Insert element e (cost Ce) into Q2 and correspondingly the reference in T2.
Phase II: Join Phase
4. Probe T1 with element e for matching elements for the join and emit the result
5. Queue the result tuple r in the output buffer B
Phase III: top-k Computation Phase
6. Aggregation operator is applied on the output buffer B to calculate the count of bids in each category
7. The bid counts are sorted to calculate top-k
Similarly repeat the steps for any element that reaches in stream S1 symmetrically with converse data structures.

6.6 Load Shedding Mechanism for Aggregation

The load shedding is implemented through the priority queues as shown in Figure 9. As aggregation is a blocking operation, we have moved it after the join, to provide join early on. The load shedding operators are added to each incoming stream before the join take place. This way, the unimportant tuples go out of the system before incurring any CPU time by taking part in joining, thus saving critical CPU cycles for processing the tuples that are relevant to the query result. This decides which tuple (node) to shed based on the weight of the node. The fast access to priority queue helps faster decision on choosing the ideal victim tuples to shed, thus acting as a very fast shedding operator. Our window is a fixed size hash buffer, one for each source stream. The data is kept in the form of a heap data structure of elements sorted by the cost of each element. The highest cost element (the one having highest probability of shedding) remains at the top ready to be shed. The shedding action is triggered as a complete element arrives at the source. The details of steps of processing are described in Table 2.



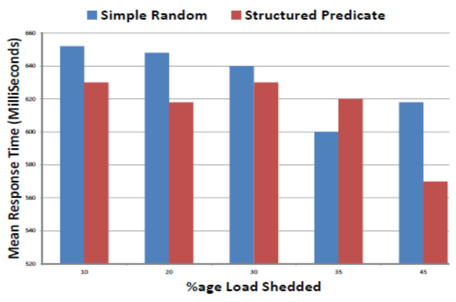


Fig. 11. Effect of Simple Random Load Shedding and Structured Predicate Load shedding on accuracy

Fig. 12. Effect of Simple Random Load Shedding ar Structured Predicate Load shedding on Latency

7. Experimental Evaluation

All our frameworks have been implemented in Java and run in systems with 2-GHz Intel Core 2 Duo processor, 2.0 GB of main memory and windows XP. The load shedding framework is implemented as a pluggable module and part of the intermediate buffer; external to our core query processor.

7.1 Framework for Set-Valued Queries

We have implemented both syntactic Simple Random and semantic Structured Predicate load shedding techniques. 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 Figure 11 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. 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 Figure 11, 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 Figure 12, 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.

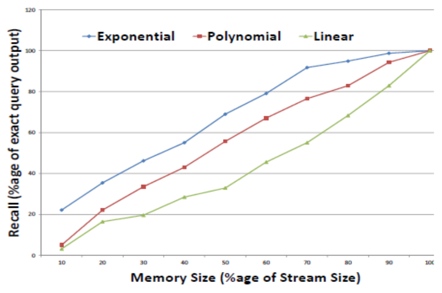


Fig. 13. Effect of Cost Function on Productivity for synthetic XMark data

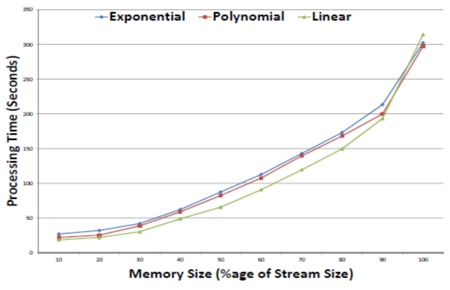


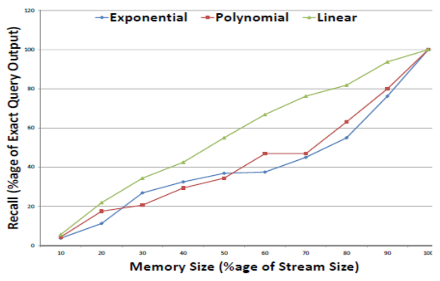
Fig. 14. Effect of Cost Function on Processing Time for synthetic XMark data

7.2 Framework for Join Query

We compare the results of our join query implementation with that from two other systems (Stylus Studio [1] and SQL Server 2005 [54]) that deal with non-stream data. Our comparison is mostly based on three factors. One, the overall quality of the result, known as recall or productivity. Second, the overall memory consumption by system. Third, the time of processing. We tested three different implementations of our cost function (linear, polynomial and exponential) for the test. Furthermore, our experimental results reveal several other properties and characteristics of our embedding scheme with interesting implications for its potential use in practice. We tested our framework on representative datasets derived from synthetic and real-life datasets. The size of the dataset is controlled to avoid the memory limitations of the systems used [1, 54].

Synthetic Data Sets We used the synthetic data from XMark [59] XML data benchmark that is modeled on the activities of an on-line auction site (www.xml-benchmark.org/). We controlled the size of the XMark data to 50 MB using the scaling factor input to the data generator. The ceiling of 50 MB is considered due to the Java heap space for Stylus Studio. We ran a set of join queries similar to the following one on all three implementations (ours, Stylus Studio and SQL Server 2005). The queries have been modified to suit the implementation [54].

Effect of Decay Our first set of experiment is to see the effect of decay algorithm on the productivity. We calculated the number of output tuples for each of the three algorithms and compared it with the exact query recall to compute the accuracy. They are measured for each of the memory size. The 100% of memory size refers to no load shedding having buffer equal to the size of the stream; in our case of 50 MB for each stream. The other memory sizes are reduced according to the ratio. Figure 13 shows the relative quality of the result for different implementations of the load shedding mechanisms. The Exponential Decay based cost function produces better result for almost all memory sizes. The stream characteristic best suits the decay function. Figure 14 indicates the better processing time for Linear Decay based cost function relative to other two implementations due to less maintenance overhead of cost calculation for shedding. As the amount of shedding decreases for higher memory sizes the gap narrows down.



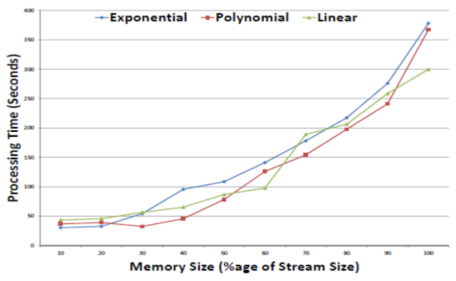


Fig. 15. Effect of Cost Function on Productivity for real life DBLP data

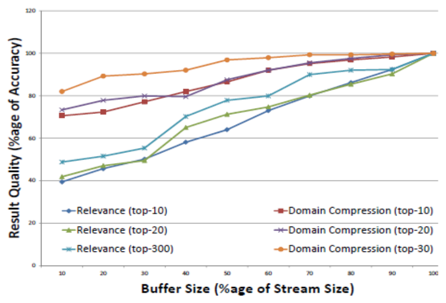
Fig. 16. Effect of Cost Function on Processing Time for real life DBLP data

Real Life Data Sets We use the real life XML data sets, DBLP [48] for the following queries. We fragment it into two parts DBLP1 and DBLP2 based on journal and conference series [44]. We adjusted the _le size to 50 MB each to make the joining load even. Once again we ran it through all five systems; three of our own implementation, Stylus Studio and SQL Server 2005. On this dataset, we use the following XQuery template that asks for authors that have published in at least 2 different journals and/or conference series:

```
for     $a1 in doc('dblp1.xml')//author,
$a2 in doc('dblp2.xml')//author
where   $a1/text() = $a2/text()
return  $a1
```

We plotted the results for accuracy for all of our three implementations. The recall percentage is calculated with respect to the output that is acquired from SQL Server 2005. The result is presented in Figure 15. However as the data is no more dependent on the time or not temporal in nature, the type of decay cost function has relatively less effect on the recall. Rather the linear function has better effect compared to other two implementations due to its simpler implementation. The high fan out characteristic of DBLP might have contributed to this implementation.

The processing time is calculated for various cost function implementations for all ten memory sizes and plotted in Figure 16. It is quite clear that the linear implementation provides better timing relative to other two implementations. Combining both the recall study and the processing time, it is evident that the linear costing function based load shedding strategy is the best one out of the three for dblp dataset.



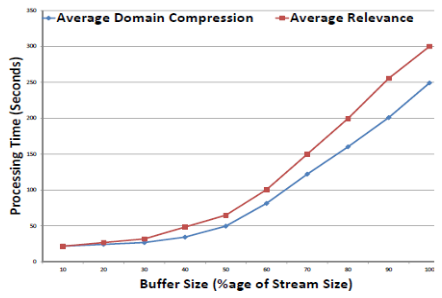


Fig. 17. Effect of Domain Compression and Relevance Shedding on result quality

Fig. 18. Effect of Domain Compression and Relevance Shedding on Processing Time

7.3 Framework for Aggregation Query

We compare the results with our earlier implementation [26] that uses the relevance index as shedding mechanism and with exact result using Stylus Studio. Our comparison is mostly based on three factors; one, the overall quality of the result, second, the overall memory consumption by system and third, the time of processing. We tested our implementation with 3 different `k' values (10, 20, and 30). Our experimental results reveal several other properties and characteristics of our embedding scheme with interesting implications for its potential use in practice.

We used the synthetic XMark [59] XML benchmark data of 50 MB size, using the scaling factor input to the data generator. The ceiling of 50 MB is considered due to the Java heap space for Stylus Studio. We ran five different set of aggregation queries for different top-K values on all three implementations (our present scheme, implementation of relevance index[26] and Stylus Studio). The queries have been modified to suit the implementation [58] to calculate both hot list in each category and hot categories. Instead of calculating the hot list over a time span like [67], we ran it for entire length of stream. The definition of hot list as assumed by [67] is based on the number of bids an item receives rather than number of items being sold actually.

Effect of Shedding vs. Compression Our first set of experiment is to see the effect of domain compression compared with the relevance index [26] on the quality of the result. We calculated the top-10, top-20 and top-30 items using both the frameworks and compared them with that of accurate result obtained from stylus studio. They are measured for each of the memory size. The 100% of memory size refers to no load shedding having buffer equal to the size of the stream. The other memory sizes are reduced according to the ratio. Figure 17 shows the relative quality of the result for different implementations of the load shedding mechanisms.

The Domain Compression based cost function produces better result for almost all memory sizes. But the impact of buffer size is quite dramatic on relevance based algorithm compared to the domain compression algorithm in higher buffer sizes. Figure 18 indicates the better processing time for domain compression methodology compared to relevance based functions due to less overhead of calculation of indexes and that of shedding.

8. CONCLUSION AND FUTURE WORK

During the course of this work, we presented different load shedding frameworks suited for different processing types in XML stream. However, the central philosophy is to develop a load shedding strategy that is least intrusive from processing point of view and least resource consuming. Our load shedding layer is external to the query operator network and proactive in nature. We have introduced a logical window model that is different from established sliding window concept and spans over the entire history of the data stream. Various synopses that we use for our different models are intelligent and effective from their construction and maintenance point of view.

The author has planned to extend this work of load shedding in XML stream processing to following areas.
Systems that deals both streaming and static data - As the requirements are different for streaming and static data processing, it will be interesting to develop a system that integrates streaming data with static data transparently where the static data plays a complementary role to produce more qualitative results or help reduce the amount of data to process by influencing the processing selectivity of stream data.

Integration with commercial CEP systems - Our stream processing system can play a role of a core query processing system feeding result into event notification layer of any commercial Complex Event Processing System. Though load shedding is not followed in event processing, our framework can play a complementing role from QoS perspective as most CEPs lack the QoS implementation. Also our load shedding framework can influence or contribute to the event consumption modes of any CEP that drop or discard any events. Our time based relevance model can play a crucial role in reducing temporal foot print of events in a event processing system. As the consumption modes are more like a logical window, than a physical window, our synopsis driven relevance model is a good fit for it.

Stream Cube -We are planning to extend the implementation of our stream cube for XML data streams.

References

[1] Stylus studio - xml editor, xml data integration, xml tools, web services and xquery.
[2] Stream: Stanford stream data management (stream) project, 2003.
[3] D. Abadi. Aurora: A new model and architecture for data stream management. VLDB Journal, (12(2)):120-139, 2003.
[4] C. Aggarwal, J. Han, and P. Wang, J.and Yu. A framework for clustering evolving data streams. In Proc. VLDB Conference, 2003.
[5] C. C. Aggarwal. Data Streams: Models and Algorithms (Advances in Database Systems). Springer-Verlag New York, Inc., Secaucus, NJ, 2006.
[6] C. C. Aggarwal and P. S. Yu. A SURVEY OF SYNOPSIS CONSTRUCTION IN DATA STREAMS. Springer-Verlag New York, Inc., 2006.
[7] N. Alon, P. Gibbons, Y. Matias, and M. Szegedy. Tracking joins and self joins in limited storage. In Proc. ACM PODS Conference, 1999.
[8] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In Proc. Thirtieth International Conference on Very Large Data Bases, pages 336-347, 2004.
[9] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems, 2004.
[10] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In Proc. Twenty-First ACM SIGACTSIGMOD-SIGART Symposium on Principles of Database Systems, pages 1-16, 2002.
[11] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In Proc. International Conference on Data Engineering, number 13(4), pages 350-361, 2004
[12] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In Proc. 21st International Conference on Data Engineering, pages 118-129, 2005.
[13] S. Babu and J. Widom. Continuous queries over data streams. In Proc. ACM-SIGMOD International Conference on Management of Data, pages 109-120, 2001.
[14] H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on aurora. VLDB Journal: Special Issue on Data Stream Processing, (13(4)):370-383, 2004.
[15] S. Bose and L. Fegaras. Xfrag: a query processing framework for fragmented xml data. In Proc. 8th International Workshop on the Web and Databases, 2005.
[16] S. Bose, L. Fegaras, D. Levine, and V. Chaluvadi. A query algebra for fragmented xml stream data. In Proc. 8th International Symposium on Database Programming Language (DBPL), 2003.
[17] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. In Proc. International Conference on Very Large Data Bases, pages 215-226, 2002.
[18] K. Chakrabarti, M. Garofalakis, R. Rastogi, and S. K. Approximate query processing with wavelets, 2001.

[19] S. Chakravarthy and Q. Jiang. Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing. Springer-Verlag New York, Inc., New York, NY, 2009.

[20] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In Proc. ICALP, 2002.

[21] S. Chaudhuri and K. Shim. Including group-by in query optimization. In Proc. 20th International Conference on Very Large Databases, pages 354-366. Morgan Kaufmann, 1994.

[22] G. Cormode and S. Muthukrishnan. What's hot and what's not: Tracking most frequent items dynamically. In Proc. ACM PODS Conference, 2003.

[23] G. Cormode, V. Shkapenyuk, D. Srivastava, and B. Xu. Forward decay: A practical time decay model for streaming systems. In ICDE, pages 138-149.

[24] A. Das, J. Gehrke, and M. Riedwald. Approximate join processing over data streams. In Proc. ACM SIGMOD, pages 40-51, 2003.

[25] R. Dash and L. Fegaras. Synopsis based load shedding in xml streams. In Proc. DataX, 2009.

[26] R. Dash and L. Fegaras. A load shedding framework for xml stream joins. In Proc. DEXA, 2010.

[27] R. Dash and L. Fegaras. A load shedding framework for processing top-k join aggregation queries. In Proc. EDBT, submitted for publication.

[28] L. Ding, N. Mehta, E. A. Rundensteiner, and G. T. Heineman. Joining punctuated streams. In Proc. EDBT, pages 587-604.

[29] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In Proc. SIGMOD, 2002.

[30] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Sketch-based multi-query processing over data streams. In Proc. EDBT Conference, 2004.

[31] L. Fegaras, R. Dash, and Y. Wang. A fully pipelined xquery processor. In Proc. XIME-P, 2006.

[32] B. Gedik, K. Wu, P. S. Yu, and L. Liu. A load shedding framework and optimizations for m-way windowed stream joins. In Proc. IEEE, 2007.

[33] L. Golab, D. DeHaan, E. Demaine, A. Lopez-Ortiz, and J. Munro. Identifying frequent items in sliding windows over on-line packet streams. 3rd ACM SIGCOMM conference on Internet measurement, pages 173-178, 2003.

[34] L. Golab and M. T. O. zsu. Processing sliding window multijoins in continuous queries over data streams. In Proc. VLDB, 2003.

[35] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In Proc. ACM STOC Conference, 2001.

[36] J. Han, Y. Chen, G. Dong, J. Pei, B. Wah, J. Wang, and D. Cai. Stream cube: An architecture for multi-dimensional analysis of data streams. (18(2)):173-197, 2005.

[37] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In Proc. 1996 ACM SIGMOD international conference on Management of data, pages 205-216, Montreal, Quebec, Canada.

[38] M. Hong, A. Demers, J. Gehrke, C. Koch, M. Riedewald, and W. White. Massively multi-query join processing. In Proc. SIGMOD, pages 11-14, Beijing, China, 2007.

[39] Q. Jiang. Data stream management system for MavHomeA Framework for Supporting Quality of Service Requirements in a Data Stream Management System. PhD thesis, The University of Texas at Arlington, 2005.

[40] Q. Jiang and S. Chakravarthy. Queueing analysis of relational operators for continuous data streams. In Proc. 12th international conference on Information and knowledge management, New Orleans, LA, USA, 2003.

[41] Q. Jiang and S. Chakravarthy. Data stream management system for mavhome. In Proc. Annual ACM SIG Symposium On Applied Computing, pages 654-655, 2004.

[42] Q. Jiang and S. Chakravarthy. A framework for supporting load shedding in data stream management systems. Tr cse-2004-19, UT Arlington, 2004.

[43] Z. Jiang, C. Luo, W. Hou, F. Yan, and Q. Zhu. Estimating aggregate join queries over data streams using discrete cosine transform. In Proc. 17th International Conference on Database and Expert Systems Applications, pages 182-192, 2006.

[44] R. A. Kader, P. Boncz, S. Manegold, and M. van Keulen. Rox: Run-time optimization of xqueries. In SIGMOD09, 2009.

[45] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In Proc. ICDE, pages 341-352, 2003.

[46] B. Kendai and S. Chakravarthy. Load shedding in mavstream: Analysis, implementation, and evaluation. In Proc. British National Conference on Databases (BNCOD), pages 100-112, 2008.

[47] Y.-N. Law and C. Zaniolo. Load shedding for window joins on multiple data streams. In Proc. The First International Workshop on Scalable Stream Processing Systems (SSPS'07), Istanbul, Turkey, 2007.

[48] M. Ley. Dblp xml records.

[49] L. Lim, M. Wang, and J. S. Vitter. Cxhist : An on-line classification-based histogram for xml string selectivity estimation. In Proceedings of the VLDB, 2005.

[50] G. Manku and R. Motwani. Approximate frequency counts over data streams. In Proc. VLDB Conference, 2002.

[51] G. Manku, S. Rajagopalan, and B. Lindsay. Random sampling for space efficient computation of order statistics in large datasets. In Proc. ACM SIGMOD, pages 100-112, 1999.

[52] D. Megginson. Simple api for xml, 2002.

[53] R. Motwani, J. Widom, A. Arasu, B. Babcock, M. D. S. Babu, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In Proc. CIDR, pages 245-256, 2003.

[54] S. Pal, I. Cseri, O. Seeliger, M. Rys, G. Schaller, W. Yu, D. Tomic, and A. Baras. Xquery implementation in a relational database system. In Proceedings of the 2005 VLDB, 2005.

[55] N. Polyzotis and M. Garofalakis. Structure and value synopsis for xml data graphs. In Proc. VLDB Conference, 2002.

[56] N. Polyzotis and M. Garofalakis. Xcluster: Synopses for structured xml content. In Proc. ICDE Conference, 2006.

[57] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Approximate xml query answers. In Proc. ACM SIGMOD Conference, 2004.

[58] S. Saha, J. Fan, N. Cohen, and R. Greenspan. The rank group join algorithm: Top-k query processing in xml data.

[59] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, , and R. Busse. Xmark: A benchmark for xml data management. Proc. VLDB, pages 974-985, 2002.

[60] S. Schmidt, R. Gemulla, and W. Lehner. Xml stream processing quality. In Proc. 1nd International XML Database Symposium (XSym), 2003.

[61] R. Schweller, A. Gupta, E. Parsons, and Y. Chen. Reversible sketches for efficient and accurate change detection over network data streams. In Proc. Internet Measurement Conference, 2004.

[62] J. Shanmugasundaram, U. Fayyad, and P. S. Bradley. Compressed data cubes for olap aggregate query approximation on continuous dimensions. In Proc. 5th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 223-232, San Diego, California, United States, August 1999.

[63] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In Proc. VLDB, 2004.

[64] N. Tatbul and S. Zdonik. Window-aware load shedding for aggregation queries over data streams. In Proc. 2006 International Conference on Very Large Databases, pages 799-810, 2006.

[65] N. Tatbul, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In Proc.VLDB, 2003.

[66] T. M. Tran and B. S. Lee. Transformation of continuous aggregation join queries over data streams. In Proc. SSTD, pages 330-347, 2007.

[67] P. Tucker, K. T. V. Papadimos, and D. Maier. Nexmark - a benchmark for queries over data streams. 2002.

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

[69] M. Wei, E. A. Rundensteiner, and M. Mani. Utility-driven load shedding in xml streams. In Proc. WWW, Beijing, China, 2008.

[70] M. Wei, E. A. Rundensteiner, and M. Mani. Achieving high output quality under limited resources through structure-based spilling in xml streams. In Proc. VLDB, number 3(1), pages 1267-1278, 2010.

[71] F. Yu and W. Shan. Compressed data cube for approximate olap query processing. J. Computer Sc. Tech, 17(5):625-635, 2002.

[72] S. Zdonik, Stonebraker, M. M., Cherniack, U. Cetintemel, M. Balazinska, and H. Balakrishnan. The aurora and medusa projects. IEEE Data Eng. Bull, (26(1)):3-10, 2003.

[73] X. Zhou, H. Thakkar, and C. Zaniolo. Unifying the processing of xml streams and relational data streams. Proc. 22nd International Conference on Data Engineering, page 50.