IMPROVING QUALITY OF SERVICE IN XML DATA STREAM PROCESSING USING LOAD SHEDDING

by

RANJAN DASH

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2012

To my Teachers and my Family

who set the example and made me who I am.

ACKNOWLEDGEMENTS

I would like to thank my supervising professor Dr. Leonidas Fegaras for constantly motivating and encouraging me to complete this thesis work. His relentless persuasions, invaluable advice and deep insight into my research have helped me immensely during the course of this doctoral study. I wish to thank my academic advisors Dr. Gautam Das, Dr. Sharma Chakravarthy, Dr. Ramez Elmasri and Mr. David Levine for their interest in my research work and for taking time to serve in my dissertation committee.

I would also like to extend my appreciation to BNSF Railways for providing financial support for my doctoral studies. I wish to thank Bonnie-Henn Pritchard, Bob Jackobson and Eldon Specht at BNSF, who made it possible. I recognize the sustained support and constant motivation from Wallace Swanson, who made me successfully complete this study. I recognize the help and encouragement from managers and team members at BNSF particularly Jade Wright and Cheryl Fernandez. I am especially grateful to my colleague, Cristian Gheorghiu for his interest in my research and for his helpful discussions and invaluable comments.

I am grateful to Indian Railways who set me in this path of higher study by giving me material support and a sabbatical. I am grateful to all my teachers in India and in United States, who taught me how to open my eyes to this world. I would like to thank Dr. Radha Mohapatra (Information Systems of Business School, UTA) for his encouragement to pursue graduate studies. I am extremely grateful to my wife, Asita and son, Akshar for their sacrifice, encouragement and patience during course

of this journey. Finally, I would like to express my deep gratitude to my parents (mine and Asita's) who have encouraged and inspired me throughout my studies. I am extremely fortunate to be so blessed. I also thank several of my friends who have helped me throughout my career.

April 20, 2012

ABSTRACT

IMPROVING QUALITY OF SERVICE IN XML DATA STREAM PROCESSING USING LOAD SHEDDING

RANJAN DASH, Ph.D.

The University of Texas at Arlington, 2012

Supervising Professor: Leonidas Fegaras

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 traffic 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, fall short of serving this new class of applications, which require 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 and processing requirements.

A common but challenging issue in DSMS, is to deal with unpredictable data arrival rate. Data arrival may be fast and bursty at times that surpass available system capability to handle. When input rates exceed system capacity, the Quality of Service (QoS) of system outputs falls below the acceptable levels. The problem of system overloading is more acute in XML data streams than its counterpart in relational streams, as XML streams have to spend extra resources on input processing

and result construction. The main focus of this thesis 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. One established method is to shed load by selectively dropping tuples under these conditions. This method helps to improve the observed latency of the results but degrades the answer quality.

In this dissertation, we first define the QoS in the context of XML stream processing and then various mechanisms to improve the QoS, specially the method of load shedding. We provide a general solution framework for implementing Load Shedding using Synopses, while minimizing the loss in result accuracy. Then, we present specific situations where issue of QoS is very critical, such as cases of aggregation and join queries. Finally, we provide techniques to handle load shedding in these cases that provide high QoS in the XML data stream systems.

In the final part of this thesis, we investigate issue of processing aggregation (group-by) join queries on data streams that provide exact results and we extend our solutions to address some of the OLAP issues in data streams.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Recently, with advancement of science and technology, data is being generated overwhelmingly from heterogeneous, disparate systems that needs to be acted upon to get information out of it. This data, which is in the form of data streams is unbounded in nature. Due to the volume and the rate of data generation, classical model of data analysis through Database Management Systems (DBMS) is no longer applicable to this form of data. Unlike transactional systems, where the data needs to be stored for future analysis, this type of data has a temporal dimension and needs to be acted upon at real-time or near real-time fashion. This data is transient, since there is no need to store and look at it again in the future, as the data changes rapidly through times. Due to this dynamic nature of data, DBMS functions, such as storing and indexing in more or less static data repository, do not apply to the stream data model. There is a paradigm shift not only in what you expect out of this data, but also on how you can process it and what tools you can use, compared to the traditional DBMS.

When data sources continuously disseminate stream data and applications continually run the so-called continuous queries on that data, the traditional query processing techniques of database systems does not directly meet the application's requirements. The processing of this type of data is normally handled by systems known as Data Stream Management Systems (DSMS). Generally, a DSMS differs from a DBMS in following three ways: First, the volume of the disseminated data may be so high that the cost for storing these data sets (even temporarily) would be

infeasible. Second, the data set may be infinite and thus cannot be stored completely. Third, query processing results are expected in real-time or near real-time, either to trigger some actions or to be fed into some applications that generate some notifications. In some applications, the results are expected to be available as soon as the first data item arrives in a DSMS. In such a scenario, 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 and is more affected by properties of input streams. Unlike a DBMS, which is mostly pull-based system, a DSMS is push-based and is vulnerable to data flow characteristics of input streams. The increase rate in network bandwidth compared to that of disk access time is one of the prime factors for explosion in data stream applications, thereby making them ubiquitous.

1.1 Data Stream Applications

The data stream applications are ubiquitous and found in increasing number. As categorized in [1], these applications can be grouped under four basic categories: Transactional, Monitoring, Causal, and Compositional. Examples of some data stream applications under each categories are covered as described below.

Transactional - *Fraud detection applications* monitor ATM and Credit Card transactions for abnormal usage behavior. *Click Streams* generated from individual's Internet usage are used to push customized advertisements and forces personalized web pages. *On-line auction applications*, such as ebay, provides a stream based platform to do auctions and bids on those auction items, thereby generating auction and bid streams. Telecommunication applications, such as *Call Detail Record* [2], can be used to generate billing information in real-time as soon as the calls are processed.

Monitoring - As the name suggests, these are data stream systems that continuously monitor events or conditions for interesting patterns and outliers, to generate notifications for remedial actions. The majority of data sources are data sensors, that generate data at specified intervals or when certain conditions occur. *Weather Monitors*, which detect extreme climatic conditions, habitat monitoring, biomedical monitoring, road traffic monitoring, RFID-based asset tracking, GPS-based location tracking, network traffic monitoring etc. are some noteworthy monitoring systems that are data stream systems. The on-line analysis over these streams includes discovering correlations, identifying trends and outliers (fraud detection), forecasting future values etc. Industries, such as Railroads rely heavily on sensors for their safe operation now-a-days. As the author is from this industry, various use cases where data stream systems are being used or can be used in Railroad context are covered in the next section.

Causal - Causal data is generated in control systems (for example, hot axle detector sensors in railroads, traffic, logistics) where interesting events are tracked to generate notifications.

Compositional - Financial analysis applications, such as *stock trading* help detecting stock rating changes that affect stock prices. *News alerts* (RSS feeds) compose news sources from various sources and push the data to interested subscribers. *Event composition* in CEP (Complex Event Processing) systems help create events from results of underlying data stream processing to generate required notifications.

1.2 Data Stream Systems in Railroad

As an internal member of a class I railroad, the author has witnessed the explosion of data stream applications in the field of railroads over last few years. Due to the very nature, the railroads are geographically spread over a large network and

hence depend heavily on the use of various sensors all over its network to monitor, collect, analyze and remedy any operational hurdles in a centralized manner. Some of the practical applications that can be classified as data stream applications and can be solved using data stream paradigm are covered next.

Detect and Process Locomotive Defects - Locomotives have programmable sensors on board that detect various mechanical defects. Some defects need to be corrected as soon as possible and others can be addressed when it is most advantageous for operational continuity. The locomotives that are operating at low horsepower are need to be identified in real-time and need to be routed to a maintenance facility. The underlying locomotive data comes in to back office in the form of an XML stream.

Warm Bearings - Hot box detectors placed along the track side read temperature readings of the warm bearings. Assume 5000 events/sec streaming in from one thousand plus detectors. The read data is to be joined with weather data at the reading location to get the relative temperature of the bearing. Bearings whose relative temperature (temperature of the bearing - ambient temperature) is greater than 180 F need to be reported for corrective actions. The system generates notifications of different degree by publishing the event with number of abnormalities.

Acoustical Detectors - They help prevent damage to tracks by identifying flat tires through sound detection.

Truck Hunting Detector - It tracks the trajectory of the truck and thereby identify any defective loading or hanging parts.

GPS - There are a number of GPS-equipped moving assets in railroad, such as locomotives, crew vans (to haul crews to and from trains), and end of train devices (ETDs) (portable devices required to be put on the back of every train to identify the end of the train). A single train may have n number of locomotives and one ETD on it. GPS data from each of these sources need to be monitored to know (1) to check

if the train is within its movement authorities (limit compliance in dispatching) (2) the integrity of trains (from relative position of GPS info from ETD and locomotives of a train), (3) geofence and tripwire events for trains to avoid collisions, (4) a train stopped event if the locomotives on the train come to a stop, (5) a crew changed event (when a crew van and a locomotive stop at the same place and the same time) and real-time crew dispatching.

Train Events - Multiple sensors in the railroad generate train arrival or train departure events in the field. They are primarily AEI (Automatic Equipment Identifier), Signal Blocks in CTC (Centralized Traffic Control) and ABS (Absolute Block Signaling) territory (Control points) and GPS (fitted in all locomotives and ETDs as covered before). The first two systems are legacy systems and the GPS is a new comer. The data streams from all these systems are need to be correlated to find accurate location of train in real-time for dispatching and train control requirements. Besides the above, there are numerous use cases in railroads where data is generated as data streams and need to be processed in real-time manner to detect unusual patterns or outliers to help prevent unsafe conditions and ensure operational safety and efficiency.

1.3 Stream Synopses as an Effective Tool in Data Streams

Data stream systems resemble to data mining problems as both problem domains require approximate results. Besides, some tools that are used in data mining field are also used in data streams. The use of various statistical data summaries known as synopses is very extensive in the fields of 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 results, approximate join size estimation, compute aggregate statistics, such as frequency

counts, quantiles and heavy hitters, change detection etc. The design characteristics of any of these data summaries are time and space efficiency, evolution sensitivity, and satisfaction of the one pass constraint of data streams. While methods, such as wavelets and sketches are complex and need novel algorithmic techniques, sampling and histograms are comparatively simple and straight-forward.

Both histograms and sampling methods can be applied to multi-dimensional data streams with ease. While sampling preserves the multidimensional representation as the original data stream, different types of histograms abstracts the behavior of more complex synopsis types, such as wavelets (Haar wavelet coefficients) and sketches. Difference in relative frequencies in equi-width histogram buckets resembles Haar coefficients of any order in a wavelet, representing the frequency distribution of a data set along any dimension. In simple terms, a histogram is the representation of data distribution across any attribute by means of dividing the attribute range into a set of ranges (buckets), and maintaining 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 [3]. So bucket design is an important consideration in histograms. There are various types of histograms, such as equi-depth, V-Optimal and the most common, equi-width histograms.

Given their simplicity, from space and time complexity standpoint, histograms can represent the original data points very effectively and can summarize any statistical characteristics very efficiently. Thus, summaries of both input data stream and output data streams can be constructed using histograms in on-line basis. Both the construction and maintenance of these histograms are relatively less tasking compared to other types of synopsis. Both structural and value synopsis of a XML data can be constructed and maintained using histograms [4, 5].

1.4 Motivation

In many ways, data stream systems are different from DBMS systems. Some prominent ones are: Data stream systems process incoming tuples continuously whereas, the DBMS depends on *save now, process later* framework. The queries are continuous or long standing in DSMS whereas, they are ad-hoc and one-time in DBMS. DSMS is known for its tolerance to approximate results, real-time or near real-time processing requirements (minimal tuple latency) and its dependency on data arrival rate etc., which are not typical of a DBMS. Delivery of QoS metrics to query processing is one of the key contrasting characteristics that makes data stream systems stand out. Though, the response time is still a parameter for DBMS query compilation and optimization, yet, it is more critical in the streaming context, where real-time results are more expected.

The common QoS parameters that are monitored in data stream systems are *Response Time*, also known as *tuple latency*, *Precision* or accuracy of the result, *Throughput* or number of output tuples per unit time, *Memory usage* or maximum memory used while processing all queries over all inputs, and *Output flow characteristics*, indicating whether the output is bursty or smooth. While managing these QoS parameters is of paramount importance in DSMS, a traditional DBMS has little support for these QoS requirements. These QoS parameters not only depend on the available system resources, such as memory and CPU capacity, but they are also affected by the input characteristics to a large extent. They are also not independent from each other. For instance, the high and variable input rate due to uncontrollable and bursty data arrival causes resource bottlenecks and performance degradation irrespective of best query optimization strategies.

Unlike relational tuples where the size of tuples are fixed, XML elements in XML stream are structurally variable in size and content. The processing of XML

stream tuples has additional complexities that adds to the the processing overhead. Sometimes conversion of these to equivalent relational tuples [6] before processing adds an extra overhead to already vulnerable system resources. Alternately, the element construction from XML stream events, such as SAX events to regular XML elements, also adds an overhead.

To overcome the overload challenge, a DSMS employs various mechanisms. It continuously evaluates the system capacity and run time optimization of queries to consume the least system resources producing best QoS results. With a fixed system capacity, adapting it to the increased system load is not an option. As approximate results are acceptable to data stream systems, dropping excess load (*load shedding*) is a viable alternative. Dropping input data to reduce tuple latency without sacrificing the result accuracy is a big 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. There is a plethora of work already done to compute the proper location of these operators, distributing the excess load to be shed among these operators and when to activate or deactivate these operators. It is beyond doubt that, these additional load shedders or drop operators add an extra overhead in planning, scheduling and activating or deactivating them to already resource starved system. The main focus of this thesis is to coming up with various frameworks that effect this excess load shedding without or with very little extra overhead for a variety of queries.

In conclusion, the real challenge in XML stream systems is to process queries with different semantic complexity while delivering guaranteed QoS under fixed system resources but unpredictable input characteristics.

1.5 Our Approach

In this thesis, we present a framework for data stream processing. We cover interesting aspects of data stream processing, such as query processing, QoS delivery through different schemes of load shedding while presenting a framework to process XQuery over XML data streams using pipeline methodology [7]. We deal with issues of guaranteeing the delivery of QoS to continuous queries. We focus our work in devising various schemes that ensure the QoS delivery under resource overload situation. We define Load Shedding as a key functionality that still is not bounded by existing methods for XML streams. We have implemented load shedding methods for various types of queries in XML data streams. Effectively, we use simple structural and value summaries to help devise these load shedding strategies [8, 9, 10]. We compare our stream synopsis construction and maintenance to other existing systems and prove their effectiveness from space and time complexity points. We cover various load shedding schemes and data structures that improve QoS. We use simple histogram techniques to create structural and value synopsis of both input and output data streams that help us delivering guaranteed QoS parameters. We use the information stored in these synopses to drive the victim selection in all load-shedding schemes. Though we prove the effectiveness of our synopsis driven load-shedding for XML data streams, we believe that, they can be applied to relational streams very effectively, as well due to their general nature.

We extend our proven methodology of using synopses to drive the load shedding to answering different types of complex queries, such as group-by and top-K queries [10, 11]. We use a different type of synopsis data structure, *Stream Cubes*, to bring an effective solution to group-by queries in multi-dimensional data streams [11].

1.6 Our Contributions

In general we make following contributions in this thesis.

- We define QoS metrics for data stream systems in general and for XML streams in particular that are critical for solving resource overload problems.
- We explain how XML stream processing systems are different from relational stream systems and why the established relational load shedding solutions are not suitable for XML streams.
- We introduce synopsis driven load shedding that uses light-weight synopses to trigger different types of load shedding: random, syntactic and semantic.
- We present a novel load shedding framework for XML streams and discuss various strategies that guarantee delivery of QoS parameters to standing queries in load shedding situation.
- We extend our synopsis driven load shedding to other query types, such as stream joins, aggregation and group-by queries and their formalisms.
- We cover pipelined XQuery processing by introducing a new type of event streams, called retarded streams, and a new framework for pull-based stream processing based on these streams.

1.7 Broad Impact

We expect that the framework presented in this thesis will serve as backbone for enacting load shedding in semi-structured data stream systems. This framework will extend itself into broader scope of delivering QoS to all stream based systems including CEP systems. This will benefit all stream based systems in general and XML stream systems in particular to handle resource overloading situations.

Our framework will also benefit those processes that take help of different synopsis structures to do different database or data stream functionalities. The innovative synopsis construction and methodology that has been presented in this thesis can be used in creating and maintaining light-weight synopsis data structures in XML streams. The same concept of synopsis construction and maintenance can be applied to relational data stream systems with little modification. Its novel structural summary construction blended with summary of data content can be used in XQuery processing systems, which forms the core of any semi-structured data stream system.

1.8 Layout of this Thesis

The rest of this thesis is organized as follows. We cover the related works in the field of QoS in data stream processing and XML streams in chapter 2. Special emphasis is given to XML data stream processing and impact of QoS on it. Related works on load shedding mechanisms, synopsis construction in data streams and multi-dimensional stream analysis are also covered in chapter 2. We briefly discussed the architectural lay out of our XML stream processing system in chapter 3. The processing model that we adopted and its various challenges are discussed in this chapter. We delineated our synopsis construction for XML streams here. In next three chapters, we formulate various load shedding strategies for various types of queries and their implementations and results there on. Chapter 4 covers random and semantic load shedding for set-valued queries. XML stream joins and QoS delivery through load shedding for it is discussed in chapter 5. Load shedding for a special case of aggregation query, top-k query is covered in chapter 6. More complex aggregation scenario such as group-by queries with joins is covered in chapter 7. In this chapter, a new type of synopsis, stream cube is introduced with its theoretical foundation to

produce exact query results. We conclude in chapter 8 with discussions regarding our future direction of research.

CHAPTER 2

RELATED WORK

There is a plethora of works already done in the field of data stream systems that address issues of quality of service. We are trying to limit the coverage of this chapter on those works that deal with quality of services in a significant manner. In doing so, we will cover works that deal with delivering quality of service in both relational and XML data streams, in separate subsections. We will discuss various works that have been done on synopsis construction and its use in data stream context. We will limit our discussion on quality of service to load shedding only. We will cover works that deal with load shedding both in relational streams and XML streams. Research works done in the area of special query processing issues, such as processing of complex stream joins and processing of aggregation queries are discussed. While doing so, we will underscore the importance of precision or accuracy of query result in the QoS context. Often times, due to acceptability of approximate query results in data stream systems, the accuracy gets overlooked. So to keep it in the forefront of the issues, we will deal with it separately with more importance through a separate parameter known as *Quality of Data* or QoD [8].

2.1 QoS in Relational Data Stream Systems

Quality of service (QoS) was identified as an important attribute of overall performance measure in data stream systems by [12], as it is implemented as an integral part of *Aurora* system through QoS monitor [12, 13, 14, 15, 16]. *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. A 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.

The Stanford Stream Data Manager (STREAM) [17, 18, 19, 20, 21, 22, 23, 24] also addresses issue of resource management, such as maximum run-time memory, in the context of constraints, such as maximum latency [20]. They emphasized the importance of quality of service in [17]. Their sampling-based load shedding to deal with resource issues is covered in [21].

MavStream [25, 26] 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 the run-time optimizer for various QoS delivery mechanisms. They have designed their QoS mechanism using queuing theory [27] and addressed the load shedding mechanism extensively [28, 29].

The Constraint Aware Adaptive Stream Processing Engine (CAPE) by Rundensteiner et al. [30] is built on the adaptation model for intra-operator execution, operator scheduling, query plan structuring or optimizing, and plan distribution. These four basic components of the CAPE system use various statistics and QoS specifications to adjust their behavior. This statistics are regularly collected from the Execution Engine at each sampling point by a QoS Inspector. They have addressed the issue of load shedding or tuple dropping in special cases, such as Join queries, by employing constraints, such as sliding windows [12, 31, 32, 23] or punctuations [33, 34], to detect and discard no-longer-needed data. They have come up

with a framework, Adaptive Selection of Scheduling Algorithms (ASSA) [35], a suite of scheduling algorithms that uses the help of QoS metrics to score various scheduling algorithms at run-time [36].

2.2 QoS for XML Stream Processing

An XML data stream is a stream of XML elements that comes over the network either in the form of XML fragments [37, 38] of varying sizes or in the form XML events such as SAX events. XML fragments can contain single elements or multiple elements. In addition to the structure and data content complexities, an XML stream has to support all streaming challenges met by relational streams. Thus, 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 can be as follows.

- tuple size (number of elements and attributes)
- tuple structure (number of recursions and REFs and IDREFs)
- tuple depth (tree depth)
- fan-in (number of edges coming out of an element)
- fan-out (number of edges coming into the element)

Similar to constraints, such as available resources (CPU, Memory etc.) and data arrival rate, the above factors of an XML data stream also impact its QoS factors. The quality of service is well researched in XML field for various dissemination services, such as publisher/subscriber networks and selective dissemination of information (SDI) systems [39]. 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. Related work [40] considers query preferences (quantitative and qualitative) similar to QoS specifications to ensure best qualitative result to queries in XML stream.

Therefore, the XML data stream systems pose more challenges compared to relational stream systems as they demand more computational resources. They not only cater to the complex structure and content characteristics of the data but also have to use more processing time to construct elements out of fragments or from SAX events [8, 9].

2.3 Synopsis in Data Stream Systems

Data streams are synonymous with large volume of data and unpredictable input rate. Therefore, data streams require more space and time resources than traditional DBMSs, and often require the use of synopsis structures such as sampling, wavelets, sketches and histograms to approximate query answers. In data stream systems, these summary structures help in reducing the cost of computation and storage, while delivering approximate results, which can be an acceptable solution for many applications.

Synopses have been successfully used in the area of query estimation [41], approximate join estimation [42, 43, 44], aggregation statistics, such as frequency counts, quantiles and heavy hitters [45, 46, 47, 48], stream mining methods, such as change detection etc.[49, 50].

Random sampling has been used for variety of stream problems, such as order statistics estimation and distinct value queries [51, 47]. Reservoir sampling, which is more inclined to data of recent history than distance history, is used to sample from a moving window [52], where as biased reservoir sampling is used as an effective technique to address queries with data from distance history with reasonable accuracy [53].

Sketches and wavelets have been successfully used in data stream query estimation. Cormode et al. in [54] have proposed a unique sketch-based summary structure

(Group-Count Sketch) to track approximate wavelet summaries for both one- and multi-dimensional data streams in a very time-efficient manner. This time-efficiency is valid for both updates and queries. Gilbert et al. [55, 56] present a general approach for building sketch-based summaries over data streams that are very space efficient to provide approximate answers to aggregate queries. Sketches have been used for second moment (for self-join estimation) and join estimation [42, 43, 44]. Dobra et al. in [43] use a sketch partitioning technique to partition the join attribute domain-space and use it to compute separate sketches of each partition that ultimately used to compute the join estimate. They have extended this concept to cover multi-query processing in [44].

Histograms have been widely used in data stream systems in many ways such as equiwidth, equi-depth and V-Optimal histograms. Guha et al. have proposed construction of V-Optimal histograms ($(1+\epsilon)$-optimal histogram) over data streams [57]. Construction of wavelet-based histograms on data streams dynamically is proposed in [58]. Both structural and value synopsis of a XML data can be constructed and maintained using histograms [4, 5].

Though all the above synopsis structures have been used successfully in various fields of data stream systems, the application of synopsis 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 [8, 9, 10].

2.4 Load Shedding Frameworks and Techniques

To guarantee the system performance against a bursty data source or a unpredictable data characteristics, adaptive data stream systems employ load shedding. Through load shedding, the systems degrade performance gracefully by shedding

unprocessed tuples. The use of statistical techniques in load shedding is well established, particularly in determining operator selectivities, which in turn helps in deciding where to place the load shedders or drop operators.

2.4.1 Load Shedding in Relational Streams

Load shedding has been implemented as part of delivering QoS to standing queries in Aurora [16] and STREAM [21]. Load shedding mechanisms in both is based on near past statistics.

The 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. Though they have done this for aggregate queries, yet the basic principle still holds well for other types of queries. For example, the metric of relative error is different for monitoring queries (set-valued queries), where the attempt is made to minimize the percentage of tuples from query answer that are missing in the approximate answer. So the key is to change the metric of relative error for other types of queries as they have different loss metrics. Special care has been taken to limit load shedding overheads incurred in placement decisions for load shedders as the cost might be prohibitive and often considered as an extra processing cost per tuple when load shedding operators are activated. Placement of load shedders can be best determined by dynamic programming.

Aurora [16] differentiates its load shedding mechanism from STREAM in the way that it is more based on quality of service or utility than user specifications. The objective is to minimize the loss in utility or overall QoS. In contrast to STREAM's random shedding (sampling based), it employs semantic load shedding that are driven by user's input on relative importance of tuples. The overall scheme is a greedy

algorithm that strives to maximize the amount of load reduced while minimizing loss of utility (QoS) [16].

Load shedding in MavStream [28, 29] 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. They have implemented the load shedder as part of the input queue, which closely matches to our placement mechanism. They have proposed four different kinds of semantic shedders in addition to the random shedder. Smallest-first, largest-first, center-first and outlier-first are all based on user input values and all strive to achieve smallest relative error in the result set. They have also addressed the placement of load shedders in the query network and distribution of load to shed among the shedders.

2.4.2 Load Shedding in XML Streams

Though XML streams are similar to relational streams in principle, they are different in various ways. First, the data complexity makes XML more resource-intensive to process. Second, the construction of an XML result needs extra restructuring and assembly which requires more CPU cycles and buffer space. Third, automata-based [59, 60] query processing engines for XML data streams are quite different in nature from relational stream processing engines. Wei et al. [61, 40] have proposed a framework that enacts load shedding through a user-based preference model. It collects user preferences to rewrite XQueries into shed queries and embeds load shedding in the automaton. These strategies require extra computation for 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 [62]. Data is spilled into secondary memory when it arrives faster than it can be processed, which can be used later to reconstruct

the result when CPU idles. The concept is similar to *Archive-Metric* introduced in [63].

2.4.3 Load Shedding in Stream Joins

Joining of two or more streams could 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. [33] have prescribed punctuation-exploiting Window Joins (PWJoin) to produce qualitative result with limited memory space. An age-based model has been introduced in [64] by approximating a sliding-window join over a data stream. For a single two-way join with a fixed memory constraints, they have solved the max-subset problem under the age-based model and the sampling problem under both the frequency and age-based models. They have also extended this model to process multiple two-way joins with an overall memory constraint.

Joining streams under resource constraints has been addressed for relational streams [65, 66, 64, 63] using windowed joins. The solution through Load shedding in relational stream joins has been given by [65, 64], through randomized sketches in [66]. Kang et al. study the load shedding of binary sliding window joins with an objective of maximizing the number of output tuples [32]. Multi-join algorithms with join order heuristics has been proposed by [67] based on processing cost per unit time. The concept of load shedding in case of multiple pair-wise joins is discussed in [66]. It uses a light sketching technique to estimate the productivity of each incoming tuple and thereby making decisions to shed or keep based on it. Their methods of MSketch, MSketch-RS and MSketch*Age have a substantial productivity gain over other load shedding methods, such as Random and Aging methods. The problem of memory-limited approximation of sliding-window joins for relational streams has been

discussed in [64]. They have proposed an age-based model that addresses the max-subset problem and a random sample method on join results for aggregate queries.

Joining XML streams for a publisher/subscriber systems is proposed through Massively Multi-Query Join Processing technique [68]. It covers the problem of processing a large number of XML stream queries involving value joins over multiple XML streams and documents.

2.5 Multi-Dimensional Data Stream Aggregations

Aggregation queries, including group-bys, are more complex than other stream operations since they require unbounded memory, and thus, they are of blocking operations. They have been thoroughly studied by Arasu et al. in [22], which classifies the problem into various aggregation function bases: distributive, algebraic, and holistic. The work uses various operators based on concept of sliding windows, such as aggregation over a sliding window (ASW) and SubStream Filters (SSF) etc, by dividing the data into smaller sets. This division allows the system to perform the required aggregation function under limited memory space. Processing complex aggregate SQL queries over continuous data streams with limited memory that yields approximate answers is presented in [43]. 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.

In [69], R. Zhang et al have suggested ways to exploit situations where the different aggregate queries differ only in their choice of grouping attributes. They have proposed a method for computing and maintaining fine-granularity aggregation queries (phantoms) to limit the use of space. A load shedding method that uses a drop operator for aggregation queries over data streams is described in [70]. It uses

drop operators called Window Drop that is aware of the window properties to effect its load shedding.

Aggregation Join queries are another group of interesting queries over data streams to solve. They found their application in many situations, where data from multiple streams have to be joined and aggregated to draw conclusion. Relative location of aggregation vs. join in a query plan is an interesting problem. Many aspects of aggregation Join queries have been covered in [71, 72, 73]. Query optimization and query transformation of continuous aggregation join queries on data streams is covered in [73]. They have advocated a query plan that executes aggregation before join using sliding windows. [72] proposes a framework for estimating equi-join query sizes based on the discrete cosine transform (DCT) that provides accurate representations of data distributions. In [71] authors have proposed a similar concept to push group-by operation past one or more joins and can potentially reduce the cost of query processing significantly. [74, 75] have investigated early or late aggregation in a aggregation join queries. They have put forward early aggregation as a preferred choice, so that the aggregation join queries can be rewritten to reorder the join and aggregation operators. They have applied these methods to static databases and ad-hoc queries. We have gone along the same justification and adopted the early aggregation as the better choice.

The use of data cubes in OLAP is a common methodology. In the highly quoted paper [76], authors have proposed a suitable way to materialize a data cube. They have used all possible group-bys to come up with a lattice framework to show that it is possible to materialize parts of the views, instead of materializing all views, which is too costly. They have established the dependencies among views through this lattice structure. [77] treats a special case of data cubes, the iceberg cubes, with complex measure, such as average. In [78], authors have proposed the implementation of cube

operator for multi dimensional OLAP (MOLAP). They have prescribed array-based cube implementation for MOLAP. A good survey of all cubing techniques in the area of XML data has been covered in the form of Cubing algorithms for XML data in [79].

Stream cubes have been used in data stream systems to answer analytical questions similar to the use of data cubes in databases. [80, 81, 82] have proposed stream cubes and their implementation to perform on-line multi-level, multidimensional analytical processing on stream data. They have advocated a tilted time frame model as a multi-resolution model to register time-related data. The model registers more recent data at finer resolution whereas the more distant data are registered at coarser resolution. Instead of materializing cuboids at all levels, they developed an efficient stream data cubing algorithm, which computes only the layers (cuboids) along a popular path and leaves the other cuboids for query-driven, on-line computation. The paper [83] proposed a dynamic data cube for data streams that manages attribute groups dynamically depending on user interest areas. [84] covers issues and challenges in OLAP systems for data streams. Similar to [85], they have addressed the issue of dimensionality of data streams and proposed ways to tame the multidimensionality of real-life data streams in OLAP analysis/mining tasks. Their Cube-based Acquisition model for Multidimensional Streams (CAMS) sets the stage for dimension flattening through a suitable example. Against the backdrop of infinite size of data streams, the stream cube compression is an interesting challenge that may save space and time. Stream cube compression is covered in [86] through materializing certain cells in in-memory stream cubes. Data cube compression through suitable density distribution of data in various dimensions and aggregation functions for data which is continuous is covered in [87].

2.6 Contribution of this Thesis Compared to Related Works

The main focus of this thesis is the development of a comprehensive XML stream processing framework that processes continuous XQueries over streamed XML while addressing various stream processing issues, such as delivering QoS, handling system overloading issues through load shedding, and answering complex queries such as stream aggregation. Specifically our work makes the following contributions:

1. We have delivered a Fully pipelined XQuery processor for XML stream processing.

2. We have developed an effective XML stream load shedder that maximizes the QoS and QoD. We have developed histogram-based concise structural and value synopsis of both the incoming data stream and the output result stream. Based on this summary structure, we have implemented *Syntactic* and *Semantic* load shedding techniques for XML streams. This load shedding is not only effective in delivering QoS, but it is also user-transparent and keeps the query transformation as simple as possible.

3. We have developed a relevance based data stream model with the help of an intelligent stream synopsis that extends a logical join window over the entire stream. In the context of building this synopsis that helps us in delivering best QoS in load shedding XML stream joins, we have developed a framework for age-based and utility-based relevance. We have extended our statistical-based load shedding techniques to further complex situations, such as aggregation join queries on XML streams.

5. We have forayed into developing a novel query processing model for aggregation join queries in data streams that is *non-window* based. We applied the concept of *stream cubes* in bringing in a cost effective solution to this type of queries. Though this work is limited to relational data streams, we plan to extend this to XML streams in future.

CHAPTER 3

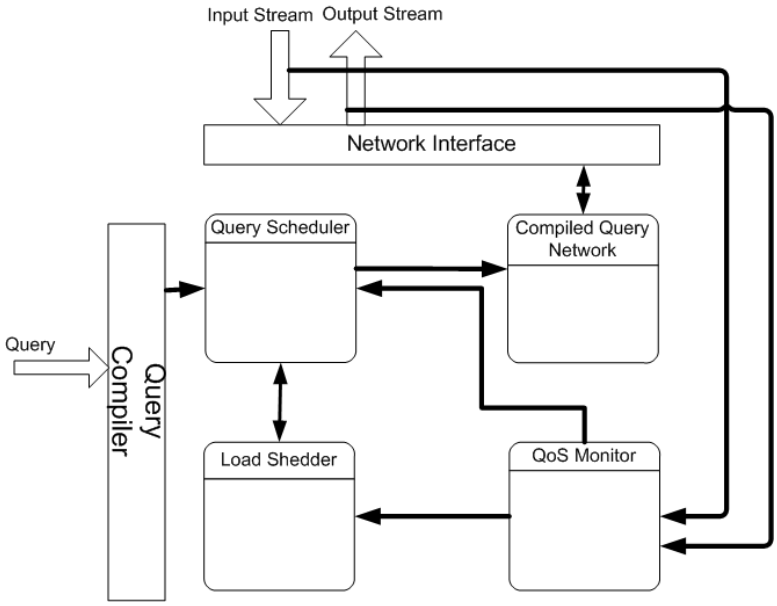
SYSTEM OVERVIEW

In this chapter, we outline our XML data stream processing system that we use for all our works. Our implementation of load shedding layer is an overlay on the core query processing engine. We also overview our synopsis construction for XML streams that forms the foundation of our load shedding framework that we cover in the next three chapters.

3.1 System Architecture

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. Unlike the relational stream processing model, we compile the standing or 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 3.1 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, those are directly influenced by our framework are as follows.



Overall XML Stream Processing Architecture

Figure 3.1. Overall XML Stream Processing Architecture.

QoS Monitor - This subsystem measures the necessary *QoS* parameters on the fly against the QoS requirements of the system. Once the QoS metrics specified in the system configuration, the *QoS Monitor* stores these and updates various parameters that influence them continuously from input and output. The *QoS Monitor* continuously monitors input/output rates, the rate of the build up of the events in the queue (intermediate buffer), and the rate at which they are consumed by the processing subsystem. There are triggers built into this monitor that control both the Query Scheduler and the Load Shedder.

Load Shedder - The Shedder uses feedback from QoS Monitor to initiate and adjust load shedding. The input streams pass through an intermediate buffer that acts as a platform for our 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, such as current load, headroom, and the necessary QoS parameters of the system, and calculates how much load to shed and in what way. The QoS specifications of the queries determine which methods to follow.

3.2 Data Model

XML data often modeled as a node-labeled tree, similar to the *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 [88]. Each event is modeled as a quadruple of the form *(i; t; l; al)*. (1) The i denotes the stream identifier, which uniquely identifies the source stream in case of multiple stream sources. (2) t is the tag or the name of the element associated with the SAX event. (3) l is the level or depth of the element in the document tree. (4) The attribute list for the element *al*, 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 have a null list. A sample XML document is shown in Table 3.1 and its corresponding events in Figure 3.2. The equivalent SAX events are shown in Table 3.2 and these events correspond to pre-order traversal of the DOM tree.

3.3 Challenges of XML Streams

The arrival rate of these events is not controlled by our system to reflect the real-world situation. The major challenge in XML stream processing system is the irregular size and form of the tuple. Unlike relational stream,s where tuples are considered as basic data units, our system does a small preprocessing to convert these stream of events into equivalent regular XML nodes. We also take cognizance of the

Table 3.1. Sample DBLP XML fragment

```
<dblp>
   <inproceedings key="conf/sigir/RoddenBSW99">
      <ee>db/conf/sigir/RoddenBSW99.html</ee>
      <author>Kerry Rodden</author>
      <author>Wojciech Basalaj</author>
      <author>David Sinclair</author>
      <author>Kenneth R. Wood</author>
      <title>
       Evaluating a Visualisation of Image Similarity(poster abstract).
      </title>
      <pages>275-276</pages>
      <cdrom>SIGIR1999/P275.pdf</cdrom>
      <year>1999</year>
      <crossref>conf/sigir/99</crossref>
      <booktitle>SIGIR</booktitle>
      <ee>
       http://www.acm.org/pubs/citations/proceedings/
       ir/312624/p275-rodden/
      </ee>
      <url>db/conf/sigir/sigir99.html#RoddenBSW99</url>
   </inproceedings>

</dblp>
```

fact that the XML data stream can exist in coarser form, such as XML fragments. However, in this case, the preprocessing of events is still needed. But instead of stitching events into elements, fragments are shred into regular elements. As we convert events into elements, we give each element a unique id that helps us in making the structural summary of the stream, which plays a critical role in our framework. Our major operations take place outside the boundary of the query processing. As shown in Figure 4.2 in Chapter 4, our processing are outside from the core query processing or after that.

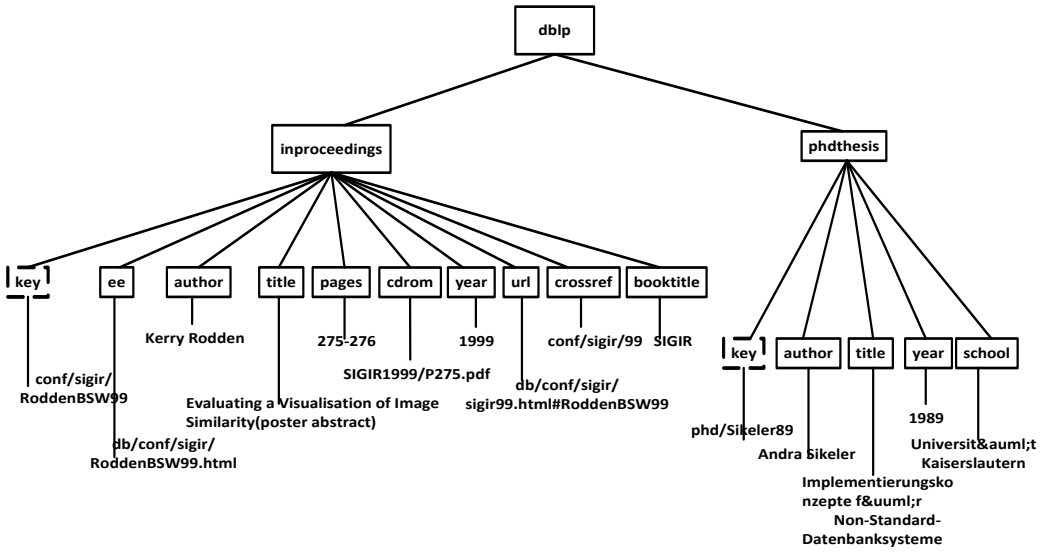


Figure 3.2. DOM Representation of XML Document in Table 3.1.

So the challenge in our system is to limit this overhead so that the functioning of the system is least affected at critical time when load shedding is enacted. The algorithms and data structures that we use are more critical from resource point of view both in space and processing time.

XML stream systems suffer from similar drawbacks as relational stream systems, but the handling of XML streams is more complex due to (1) their Hierarchical and semi-structured nature, (2) the Varying XML element size and depth, (3) the varying

Table 3.2. SAX Event Stream of Table 3.1

EVENTS	NAME	ATTRIBUTES
START DOCUMENT		
begin	dblp	
begin	inproceedings	key=“conf/sigir/RoddenBSW99”
begin	ee	
text	ee	
end	ee	
begin	author	
text	author	
end	author	
begin	title	
text	title	
end	title	
..	..	
..	..	
end	inproceedings	
end	dblp	
END DOCUMENT		

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 problem is to convert the XML stream into relational tuples before processing them. If load shedding is required, then relational load shedding techniques can be applied to this framework easily. The advantage of converting the problem into a relational stream problem is the availability of matured and well-established relational solutions to solve the problem. On the other hand, this method adds additional overhead and complexity by adding to the requirements of CPU time and increases the latency.

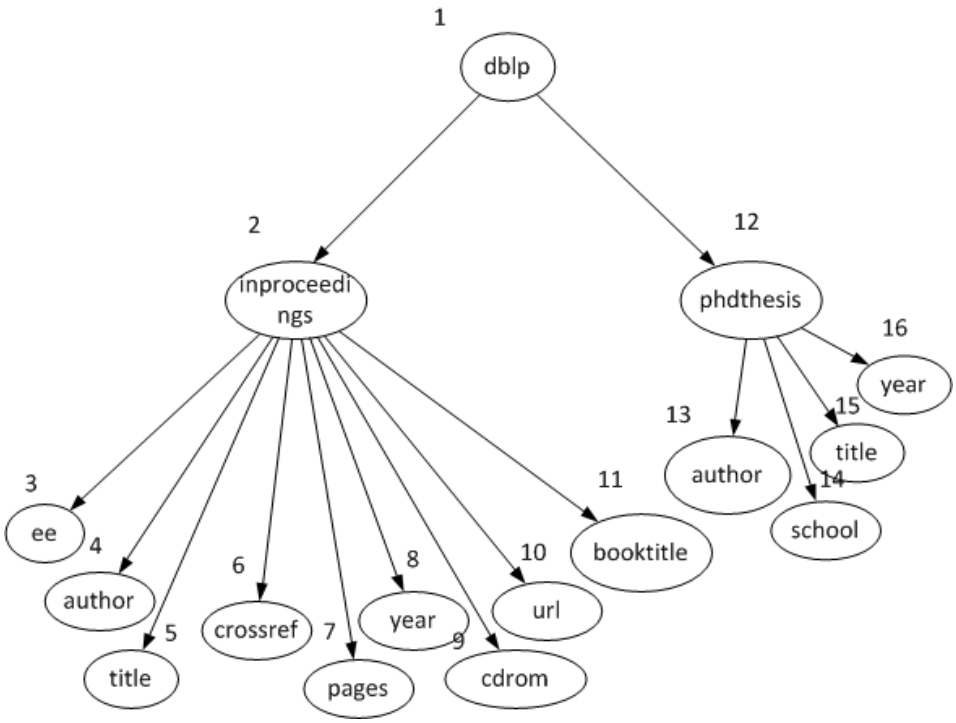


Figure 3.3. Structural Summary of XML Document in Table 3.1.

3.4 Synopsis for XML Streams

As described in Chapter 2, various types of synopses (sampling, wavelets, histograms and sketches) have been used for various problems in stream systems. The synopsis construction in these systems has to satisfy certain resource critical conditions, such as bounded storage, minimal per record processing time, and single pass criteria. Of these synopses, 1-d histograms are the most basic types that approximate the frequency distribution of an attribute value. Though these types can not capture the inter-attribute correlations, given their super linear space complexity, they can 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

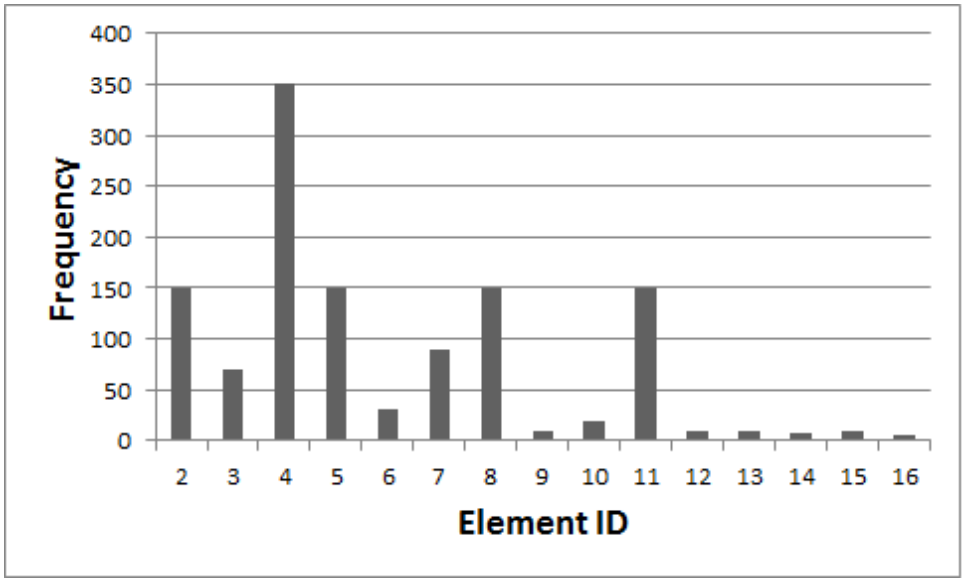


Figure 3.4. Sample Structural Synopsis of XML Document in Table 3.1.

tools. We have used the equi-width histograms in all our frameworks. This not only gives us a clear summary of data distribution when a bigger picture is needed, but they are also very simple to construct and maintain and pose the least overhead from resource standpoint. 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 attempt to limit this error by limiting 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.

3.4.1 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, which we call value and structural synopsis respectively. 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 XML document shown in Table 3.1 are shown in Figures 3.3 and 3.4 respectively. Each node or element in the tree carries its corresponding tag name and an unique id.

3.4.2 Value Synopsis

Similar to synopsis for an attribute value in a 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 document in Table 3.1 is shown in Figure 3.5.

3.5 Summary

The XML stream processing system presented in this chapter forms the core processing engine on the top of which we have built our load shedding frameworks. The synopsis construction methodology that we describe in this section is our general tool of choice that we use to spear head the load shedding frameworks for various types of queries that we cover in subsequent chapters.

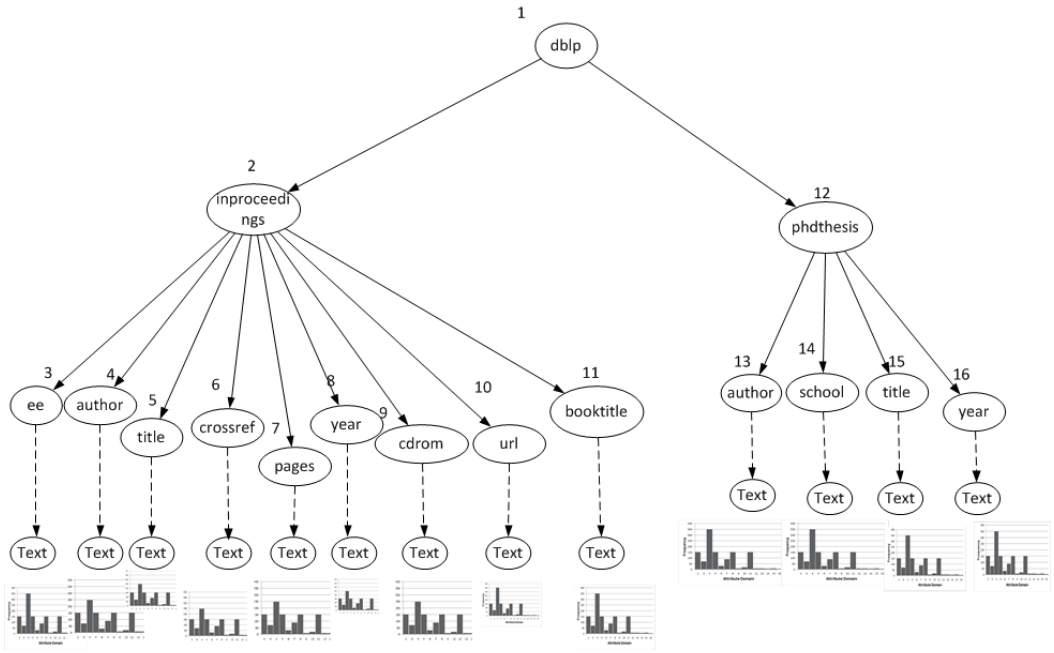


Figure 3.5. Sample Value Synopsis of XML Document in Table 3.1.

CHAPTER 4

LOAD SHEDDING IN XML STREAMS

4.1 Overview

In this chapter, we give a formal definition to our load shedding problem in XML stream system. We introduce our synopsis based load shedding in general. We also cover various types of load shedding, such as random and structure-predicate load shedding with an eye on delivering QoS to the query processing. We prove the effectiveness of our load shedding strategy by applying it to set-valued queries in this chapter. Finally, we present the implementation of these load shedding techniques on our core XML stream processing system that we introduced in the previous chapter. We defer the analysis of load shedding for other types of queries, such as join, aggregation and 'group-by' queries, to later chapters.

4.2 QoS Parameters for XML Stream Processing

From a user perspective, the most common QoS parameters in a stream processing system are

Tuple Latency - The time difference between when a tuple arrives and when it is output from the system.

Throughput - Defined as the number of tuples output per unit time. A bursty throughput is inconvenient to the user.

Precision or accuracy - The number of output tuples missing from the result set calculated as an error percentage. It can also be defined in terms of a range tolerance by the user to show which range of values are less tolerant to other ranges.

From system perspective, the quality metrics are defined as

Maximum Memory Footprint - The memory consumed by the system to process stream data against the user queries.

Processing Footprint - The gracefulness of the system while managing the required resources to achieve user specific QoS metrics.

As XML streams are more complex and nested, 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 above primary independent QoS parameters are also interdependent. 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.

4.3 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 main objective of our QoS monitoring system is to measure and monitor in real-time the following key independent QoS metrics using the least possible overhead. Unlike some existing systems, where the required QoS specifications for the query are collected from user as part of the query, ours is more user-transparent, as far as the QoS is concerned. We treat all queries equal and strive to do best for all of them. Therefore, the deliverance of required QoS is user-independent and implemented as

part of the overall system.

In our framework, the *Utility Metric* plays a key role to monitor the utility of an XML element. These utility values of the elements/tuples form the basis of our load shedding mechanism, as described in later sections. 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 are implemented through a suite of innovative data structures. As described in the previous chapter, we have built both structural and value synopses for both input and output streams. The details of implementation of these synopses and their effectiveness will be covered fully in later sections. The synopses for our stream processing system are classified 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 3.4 and 3.5 respectively.

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 ($Histogram_{input}$) 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.

4.4 Load Shedding in XML Stream Processing

As noted earlier, 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 processing demand grows rapidly with the data arrival rate and the query complexity. When some of the stream sources fall into a bursty mode or wake up from a sleeping mode, the increased demand for CPU outweighs the processing capability, which results in buffering more and more events before processing with increasing latency. Long bursts also cause 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, 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 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.

The load shedding problem in XML streams, as described in [40], 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 [40] is more biased towards structure-based preferences than value-based, while our work takes into account both structure as well as value properties of complex XML streams to improve 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 [40] in another way that their shedding is implemented using an automaton, which incurs some processing before getting shedded, whereas ours is handled much before without any overhead.

As described in the previous section, 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 that 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 higher weight. As it will be explained in next section, our framework uses a Greedy approach 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.

Besides the set of synopses, our system uses a separate data structure to keep the relative importance of unique elements in sorted form for ready reference. This is bounded in memory as the number of unique elements in the tree are bounded in an XML tree. On arrival of a new element, once the histograms are updated for their structure and value frequencies, the QoS monitor updates this relative importance structure by recalculating the value for that element and re-sorting it.

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 a preventive mechanism. We have implemented two different types of load shedding mechanisms, covered in the 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, while the semantic one is reactive.

The Simple Random Load Shedding is simple in implementation, arbitrarily sheds load to prevent congestion situation. The Structured Predicate takes into account the structural and value synopses 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, making the results approximate with a higher error bound. On the other hand, the Structured Predicate Load Shedding takes both structural and value synopses 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 one by dropping only irrelevant data and thereby producing more qualitative results. It takes into account complex workload information, and structural and value synopses to decide how much and which data 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.

The existing load shedding systems treat the problem of shedding as a network optimization problem [16, 28]. Their solution attempts to place the load shedders inside the operator network to achieve the optimum saving in resources and to distribute the load to be shed among these operators. In contrast, our system takes a holistic approach and formalizes it as an optimization problem, deciding primarily which tuple to shed. Our load shedding solution is based on the admission control

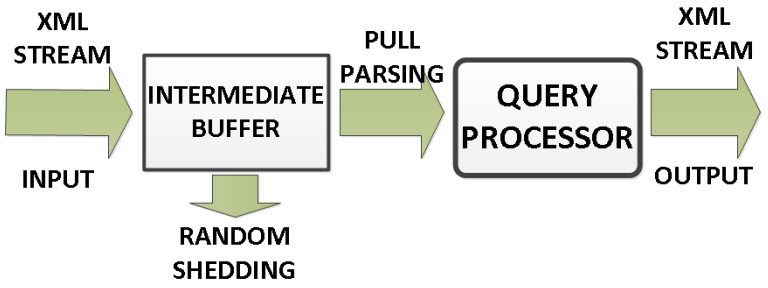


Figure 4.1. Random Shedder Implementation in Intermediate Buffer.

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 implementation 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 build up. Besides the above parameters, the system also monitors the rate of input, the rate of build-up of the buffer level, and the rate at which the events are being consumed from this buffer.

4.4.1 The Simple Random Load Shedder

As shown in Figure 4.1, 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. In the worst case scenario, the stream may be reporting at its deepest level when the trigger comes into action. Thus, 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 a 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 approximation 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 [16]. 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.

4.4.2 The 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 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 [4], structural and value synopses can be constructed for any XML data. Our load shedding framework has two main parts. The first is the construction of the appropriate structural and value summary, and the second is the

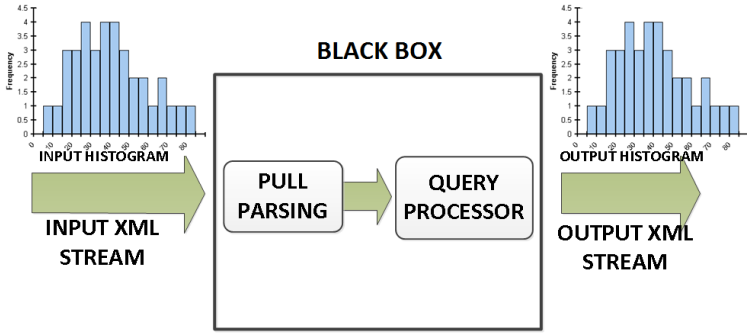


Figure 4.2. The Structured Predicate Load Shedder Implementation through a Histogram.

algorithm that performs the shedding based on this summary. There are various methods to maintain a structure-only or a value-structure summary [89, 90, 91, 92, 93, 4, 94, 95]. We chose to maintain a histogram similar to [96] 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. Our system also builds separate histograms, one for each leaf node using the leaf value (Figure 3.5).

As an example, a sample XML data tree, shown in Figure 3.3, contains dblp data [97]. The structural information of the tree is modeled by assigning unique identifiers or object identifiers to various nodes, such as dblp-1, inproceedings-2, ee-3, author-4, title-5 etc. Similarly the value of 'title', 'year', 'booktitle' etc. are hashed and mapped

onto separate value histograms as shown in Figure 3.5. 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 (Figure 4.3).

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., v_i/w_i 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. This procedure continues until the knapsack is full (the shedding requirement is achieved).

Theorem 4.4.1. *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 ($1 \leq i \leq n$) in the XML stream besides the root element and they are ordered by their relative importance (RI).

$f_{o1}/f_{i1} \leq f_{o2}/f_{i2} \leq .. \leq f_{on}/f_{in}$ where f_{oi} and f_{ii} are output and input frequency

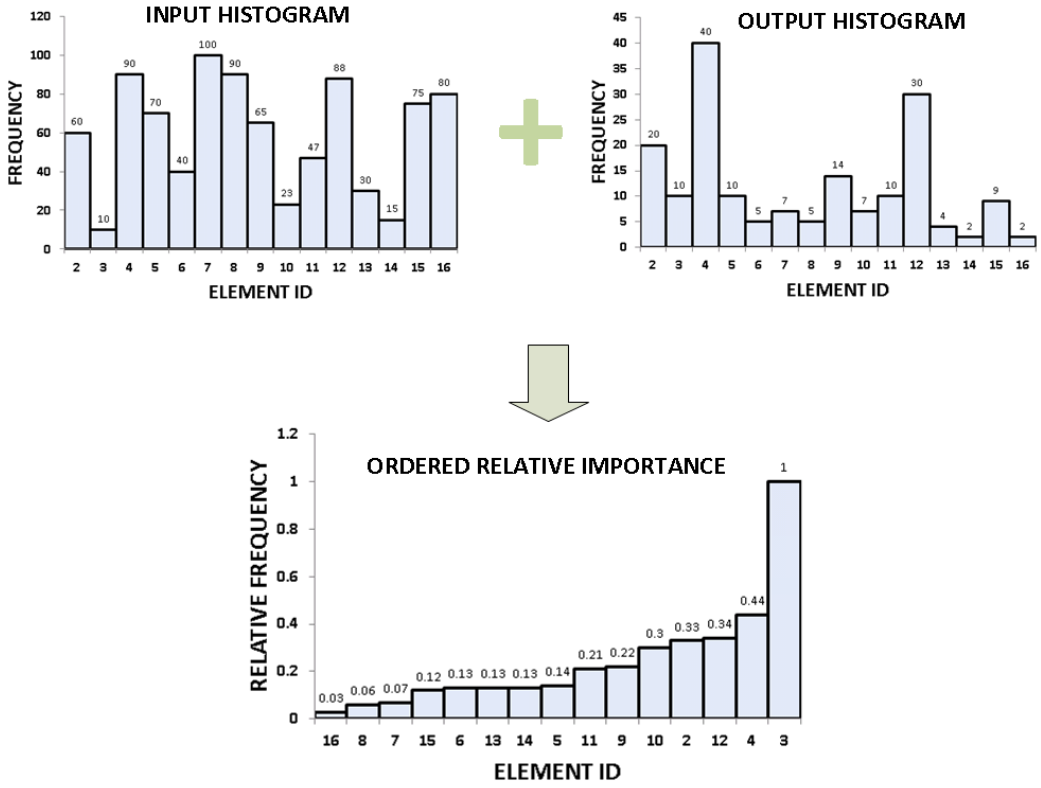


Figure 4.3. Relative Importance Construction.

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 while 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 (V) due to Y is always greater than quality loss due to X.

$V(Y) - V(X) \geq 0$

The total quality loss is $V(X) = \sum_{i=1}^{n} x_i(f_{oi}/f_{ii})$.

The total quality loss for Y is $V(Y) = \sum_{i=1}^{n} y_i(f_{oi}/f_{ii})$, Then

$$V(Y) - V(X) = \sum_{i=1}^{n} y_i(f_{oi}/f_{ii}) - \sum_{i=1}^{n} x_i(f_{oi}/f_{ii}) \tag{4.1}$$

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

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

Replacing this in Equation 4.1, we get

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

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 be shed is met.

4.5 Experimental Evaluation

We have implemented both syntactic (*Simple Random*) and semantic (*Structured Predicate*) load shedding techniques in Java as separate modules on the top of 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 two sources and implementing both load shedding techniques. The first dataset is the XMark benchmark data of size 100MB and the second dataset is 150MB of real world data derived from railroad hot box detector data streams. The schema of respective sources are attached in Appendix A. The workload was mixed

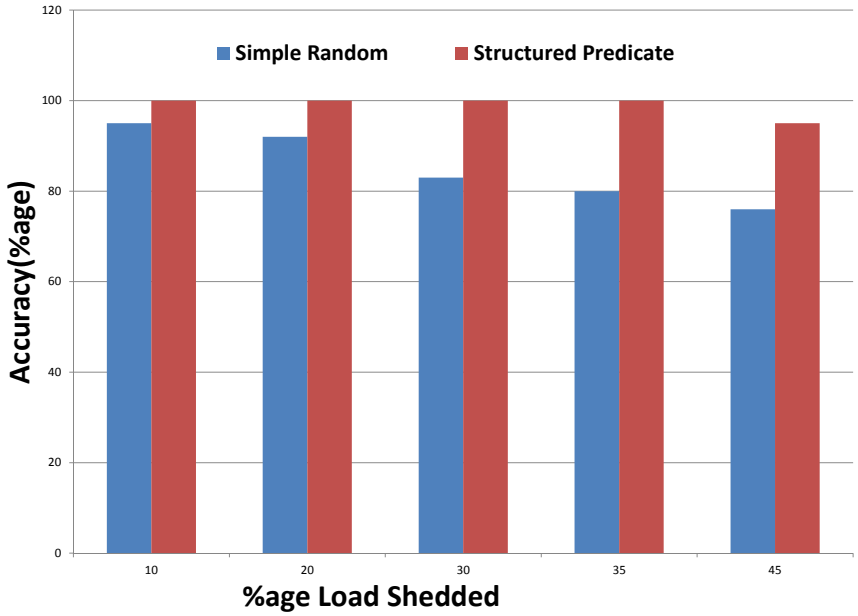


Figure 4.4. The Effect of Simple Random Load Shedding and Structured Predicate Load shedding on Accuracy.

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 4.4 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 4.3. 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 Figure 4.4., 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

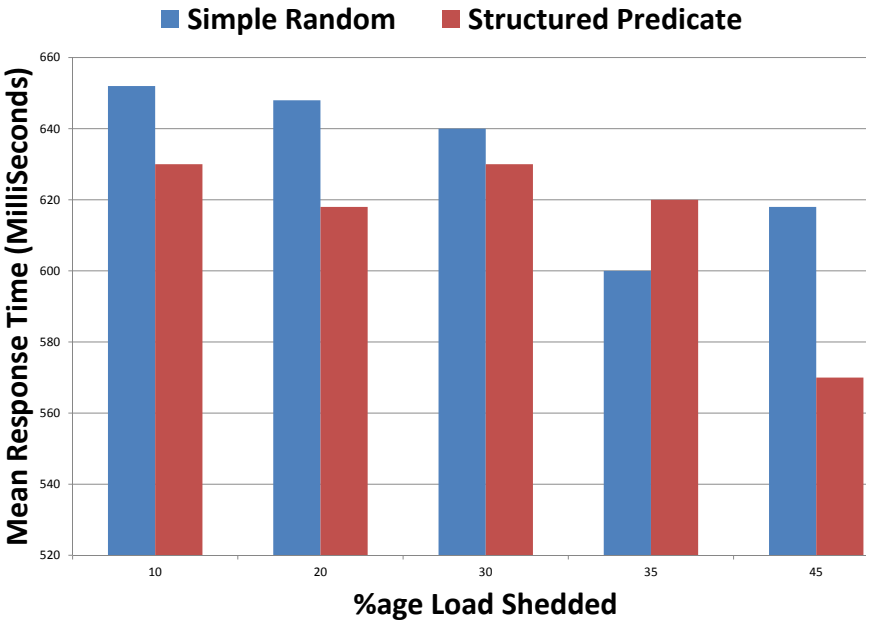


Figure 4.5. The Effect of Simple Random Load Shedding and Structured Predicate Load shedding on Latency.

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 the query result by measuring the mean response time of the query processing in milliseconds. As shown in Figure 4.5, the mean response time is always less for structured load shedding relative to the simple load shedding, except at the 35% load shedding point.

4.6 Summary

Though the relational and XML stream processing systems face similar problems due to overloading condition, but they 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. We proved the effectiveness of our strategy for set-valued queries (XPath and XQuery) and with predicates in this chapter. We formulated this strategy using innovative but simple synopses that is driven by data characteristics in both input and output. 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 performance results, we show that the semantic framework produces better result than syntactic framework. The simple load shedding, as is used in relational streams [16], leads to loss of structural integrity and violation of the schema. Load shedding in stream joins is covered in the next chapter and aggregate queries with "group-by", in the Chapter 6.

CHAPTER 5

LOAD SHEDDING IN XML STREAM JOINS

After establishing a general load shedding framework for XML streams and proving its effectiveness for set-valued queries in the last chapter, it is now time to go deeper into other types of queries, such as joins and aggregations. Usually windowing is an established method of stream processing for stream joins and aggregations. However, the limitation of window size due to memory constraints takes a heavy toll on the accuracy of the query result. In this chapter, we propose a unique windowing technique based on 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 applied our technique on XML stream domain and proved the effectiveness of our strategy through measuring the accuracy of the result from joining two XML streams using standard XQuery. With assumption of acceptability of an approximate solution with acceptable error bound in the face of unbounded, complex XML stream, we designed a low overhead architecture for load shedding and tested its usefulness through a set of cost functions. We will cover the load shedding for aggregation queries in the next chapter.

5.1 Overview

XML stream joins have been widely used in various systems, such as Publish/Subscribe systems [68], [98], message brokers in Web service-based architectures, etc. There is a need to merge various stream sources at the client end to produce

customized results. Thus, the XML streams need to be join-processed under limited resources, such as memory or CPU [99]. A comprehensive discussion on related works that cover stream joining under limited resources has been given in Section 2.4.3.

Usually the memory limitation is addressed through maintaining a continuous sliding window (*time-based* or *tuple-based*) [17] for both join input streams and applying the join between these windows. Due to the limitation on the size of the window, the result is approximate and solution paradigms, such as max subset or random samplings, are resorted to.

In this chapter, we focus on problem of load shedding for continuous queries that contain equi-joins on multiple XML streams. We strive to optimize the accuracy of the query result through an innovative method of maintaining windows that are built on the concept of relevance, rather than time or frequency, and applying the join between them. We will utilize the concept of structural and value synopses that we covered in the previous chapter in formulating the solution. We propose a framework that is built on the basis of both frequency-based and age-based model [64]. It draws its strength from both models by using a cost function that addresses the best of both families. The load shedding is driven by this cost-based element expiration time. 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.

We limit our discussions to binary joins using a fixed window that employs the shedding on arriving at steady state. The cost-based victim selection is our main process to maintain the window and thereby to ensure fixed memory utilization.

5.2 The XML Join Query Processing Model

Once again, we consider the XML stream as a stream of SAX events that needs a small pre-processing to convert into regular XML elements before running queries

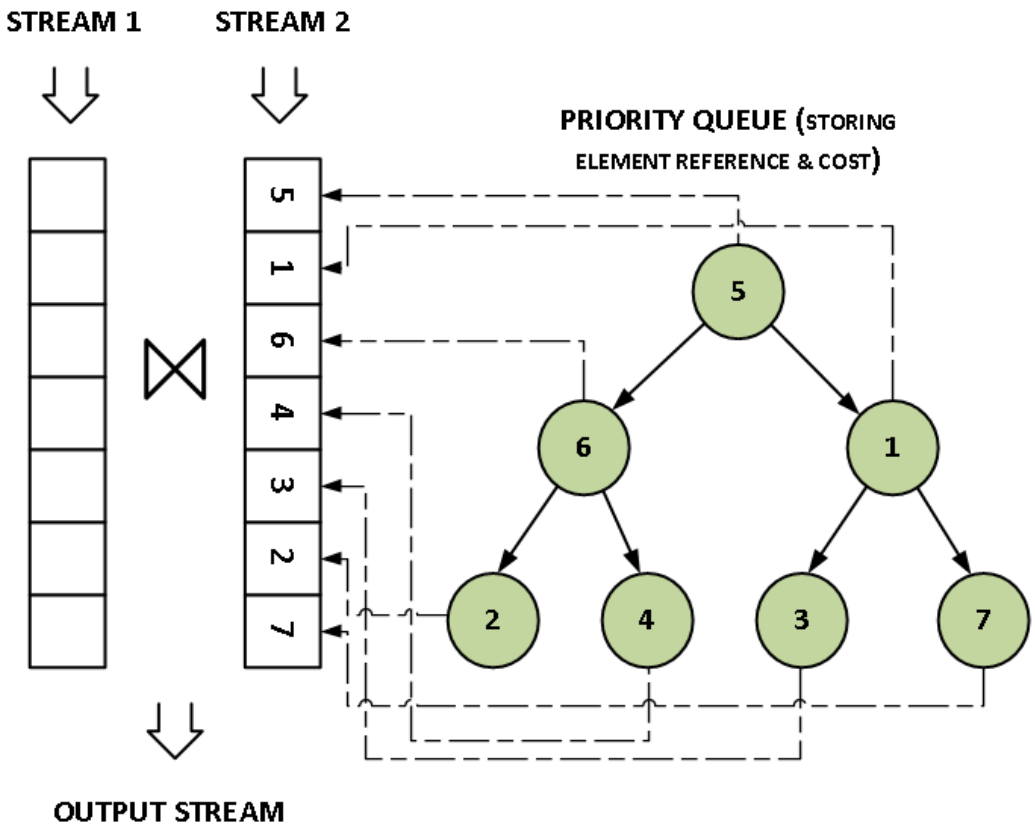


Figure 5.1. The Overall XML Stream Join Query Processing Architecture.

on them. Due to the continuous nature of a stream, our system depends on a logical window $S_i[W_i]$ that provides the necessary basis to run join queries. We limit our discussions to two streams i = 1,2. The size of the window $S_i[W_i]$ is determined by the size of the available memory. As the window is not time-based or tuple-based, it determines its content through relevance. The join query result is denoted as $T = S_1[W_1]XS_2[W_2]$.

5.2.1 Stream Models based on Frequency, Age and Relevance

In the previous chapter, we developed a relevance framework based on a frequency model. The relevance index (RI) is calculated based on the frequency of an

element in input and in output streams. However, as mentioned in [64] the frequency-based model does not work for all kinds of streams. It fails for streams where the relevance distribution is skewed over the life of the tuple, as in the case of on-line auction scenario where more bids come towards closing of an item.

Age-based models require more monitoring and frequent adjustment than the window mechanism to yield the max subset result [64]. Depending on the shape of age curve, different strategies are to be followed to optimize the result. On the other hand, if the time based window model is followed, it will result in decreased productivity due to the discard of relevant tuples when the resource is limited.

5.2.2 Relevance based Window

The solution to above issues is to optimize the productivity (max subset result) by collecting all relevant tuples while discarding the irrelevant ones. The selection will be based on a relevance framework. The relevance is decided based on its probable participation in the join process, which is calculated from a suitable mix of both age-based and frequency-based models.

To achieve the above solutions, we have resorted to a new window model. 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, thus 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. The effectiveness of the approach is measured objectively, as described in experimental evaluation section.

The window construction, though a part of the overall system, is yet an overhead that consumes valuable resources, such as memory and CPU. We have tried to make this extra layer as lean as possible with the least possible processing effort or CPU

cycle churning. In addition, we have implemented the symmetric hash join as the join mechanism to make the system more efficient, from both time and space points of view.

5.3 Synopsis for Join Processing

The main idea of this chapter is to make the best utilization of limited resources (memory, CPU) by making best judgment to prepare an intelligent synopsis for each stream that guides the load shedding decision. The synopsis construction is light and has the least overhead for its maintenance. The heart of the synopsis is the construction and maintenance of the heap based priority queue (Figure 5.1) with a relevance measure of elements. The relevance measure of each element is calculated using a cost function (Equation 5.1). Each time an element comes in any input stream, its cost is calculated and updated or inserted into the heap, based on its preexistence in the heap. The cost of all nodes gets updated at this instance too. The heap is kept updated, with the node with least value at the root, ready to be sheded if needed.

5.3.1 Cost Function

The calculation of the weight for each element in the logical queue is done using the cost function in [*Equation 5.1*]. This equation has two significant parts.

(a) *Age factor*

(b) *Utility factor*

The age component is derived from forward a decay based model [100], 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 may have lower utility for future joins,

even if it has taken part in the join at present. Therefore, depending on the type of stream these factors contribute differently towards the over all relevance. The cost of relevance for an element e_i of a stream S_i that has arrived at time t_i and measured at time t, such that $t \geq t_i$ and $t_i > L$ is

$$C = C_1 f(t_i, t) + C_2 f(u_i) \tag{5.1}$$

Where

$f(t_i, t) = \frac{g(t_i - L)}{g(t - L)}$ a decay function for an element with a time stamp t and landmark timestamp L

$f(u_i)$ = The number 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 the data type of the XML stream.

t_i = time of arrival of element e_i

t = time when the relevance of e_i is measured.

5.3.1.1 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; a weight of 1 for any element arriving at time of measuring, a weight of 0 for any element arriving at the beginning of the system start (landmark time L). The function is as follows:

$$f(t_i, t) = \frac{(t_i - L)}{(t - L)} \tag{5.2}$$

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

Polynomial Decay

$$f(t_i, t) = (t_i - L)^2/(t - L)^2 \tag{5.3}$$

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.

$$f(t_i, t) = \exp(t_i - t) \tag{5.4}$$

5.3.1.2 Utility-based Relevance

The second part of the cost function comes from the utility of each element from their participation in join operation and represents a simple form of the output history. 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 implementation. More accurate weights can be calculated based on the timestamps of its appearance in the result stream.

5.3.2 Relevance-based Window

The priority queue that is implemented as a heap acts as a logical window for the input stream. Its size is determined by the available memory and controls when to start the shedding. The victim selection is facilitated through the heap structure to shed the lowest weight element at the root. The relative weight of each element node driven by the choice of cost functions (discussed in section 5.3.1.1 and 5.3.1.2). 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.

5.4 The Load Shedding Mechanism for Join

Our basic algorithm for executing join $S_1[W_1]$ X $S_2[W_2]$ is shown in Table 5.1. If memory is limited, we need to modify the algorithm in two ways. First, in Line 2, we update$S_1[W_1]$ and $S_2[W_2]$ to free up memory occupied by expired tuples. More importantly, in Line 6, memory may be insufficient to add e to $S_1[W_1]$. In this case, we need to decide whether e is to be discarded or admitted into $S_1[W_1]$, and if it is to be admitted, which of the existing tuples is to be discarded. An algorithm that makes this decision is called a load-shedding strategy [7, 91, 65]. Due to load-shedding, only a fraction of the true result will actually be produced. We denote the fraction of the result tuples produced as *recall*.

As described above, we adopt a window-based approach to process the join query, where our window is a fixed-size buffer for each 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 if it matches with an existing element in the heap, otherwise the new element is added to the heap. 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 by the cost of the new element with respect to that of the element at the top of the heap.

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 to these elements has been maintained in the respective hash tables for faster access.

Table 5.1. Join Execution Steps

N_i: Max size of logical buffer $S_i[W_i]$
Q_i: Associated relevance queue of stream S_i
B_i: Associated hash based buffer of stream S_i
f_i: Hash functions for streams S_i. For simplicity we kept $f_1 = f_2$
$attr_i$: join attribute of stream S_i

1. When a new element e arrives in stream S_1
Phase I: Symmetric Hash Join
2. Calculate hash value of e by applying f_1 on $attr_1$ and insert in B_1.
3. Calculate hash value of e by applying f_2 on $attr_2$
4. Probe B_2 using the value from step 3.
5. Emit the result
Phase II: Synopsis Construction
6. If size of $Q_1 < N_1$:
    Insert into Q_1
  Else
    a. calculate cost of e, C_e
    b. If $C_e <$ cost of element at head of Q_1:
        throw e and remove the corresponding element from B_1
    Else
        shed head of Q_1 and Insert C_e into Q_1
(Repeat all above steps for any element that arrives in stream S_2 with related data structures.)

5.5 Experimental Evaluation

In this section, we present the results that we got from the system that we implemented based on the Section 5.4. We compare our results with two other non-stream systems, Stylus Studio [101] and SQL Server 2005 [102]. 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 the system and third, the processing time. Our implementation is in Java and we ran the systems in a 2-GHz Intel Core 2 Duo machine with 2.0 GB of main memory running windows XP. 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, 50 MB each. The size of the dataset is controlled to avoid the memory limitations of the systems used [101, 102]. We ran a set of five join queries similar to those below, on all three implementations (ours, Stylus Studio, and SQL Server 2005). The queries have been modified to suit the implementation [102].

5.5.1 Synthetic Data Sets

We used the synthetic data from XML data benchmark, XMark [103], which 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.

Sample XQuery

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $a :=
    for $t in $auction/site/closed_auctions/closed_auction
    where $t/buyer/@person = $p/@id
    return $t
return <item person="$p/name/text()">count($a)</item>
```

5.5.1.1 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 sizes. 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 a ratio. Figure 5.2 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 5.3 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.

5.5.2 Real Life Data Sets

We use the two sets of real life XML data sets; one set derived from DBLP [97] and the other from railroad data sets.

5.5.2.1 DBLP Data

We fragment it into two parts DBLP1 and DBLP2 based on journal and conference series [104]. We adjusted the file size to 50 MB each to make the joining load even. Once again, we ran it through all five systems; three of our own implementations, Stylus Studio, and SQL Server 2005. On this dataset, we use the following

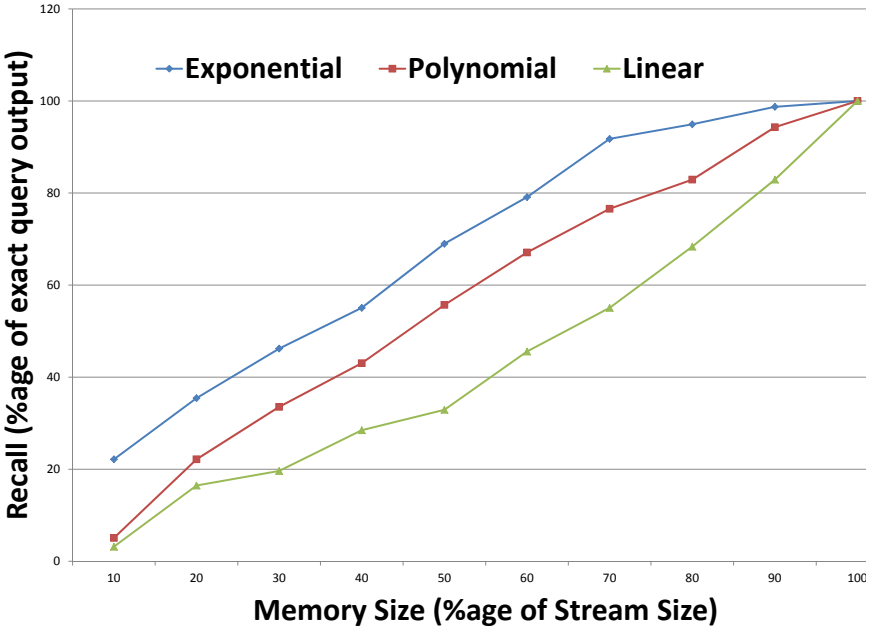


Figure 5.2. The Effect of Cost Function on Productivity for Synthetic XMark Data.

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

5.5.2.2 Railroad Data

We derive two sample data segments from AEI message stream (Appendix A.1) and OS message stream (Appendix A.2), 50 MB each to process a join based on time

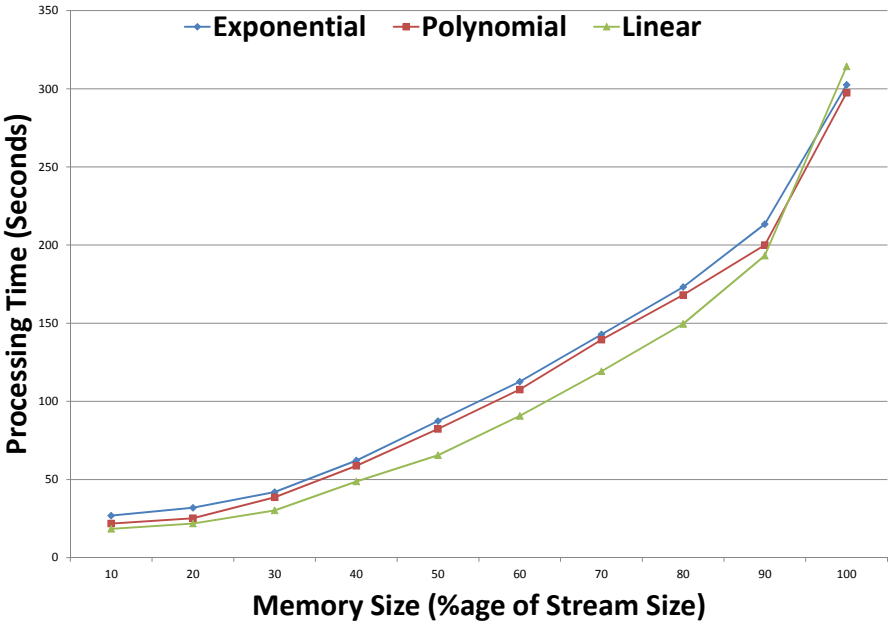


Figure 5.3. The Effect of Cost Function on Processing Time for Synthetic XMark Data.

and place of the sensor. The OS Message is not aware of the train symbol of the train that passes through it, as it is critical to dispatching limit compliance. So, the train symbol can be generated by joining the data stream from AEI with that of OS based on the place and time of data generation if the sensors happen to be at same place (following query sample) else by a time envelop and direction can be used.

```
for     $a1 in doc("AEIMessage.xml")//ConsistMsg
        $a2 in doc("OSMessage.xml")//OSMessage
where   $a1/ReadingDateTime = $a2/OsReportDateTime
        and $a1/Station333 = $a2/Station333
        and $a1/StationState = $a2/State
return  $a1/TrainSymbol
```

Similar to synthetic data, we plotted the results for accuracy for all of our three implementations for real-life data averaged from DBLP and Railroad data. The recall percentage is calculated with respect to the output that is acquired from SQL Server 2005. The result is presented in Figure 5.4. 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 processing time is calculated for various cost function implementations for all ten memory sizes and plotted in Figure 5.5. 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 load shedding strategy based the linear costing function is the best one out of the three for DBLP dataset.

5.6 Summary

We presented a load shedding strategy for stream joins and proved the effectiveness of our approach by implementing a logical window that extends over the whole stream by shedding low cost nodes. Our approach is efficient as it uses various cost functions to attenuate the data characteristics of the input stream. Though we have explained the framework using two streams, it could also be extended to commutative

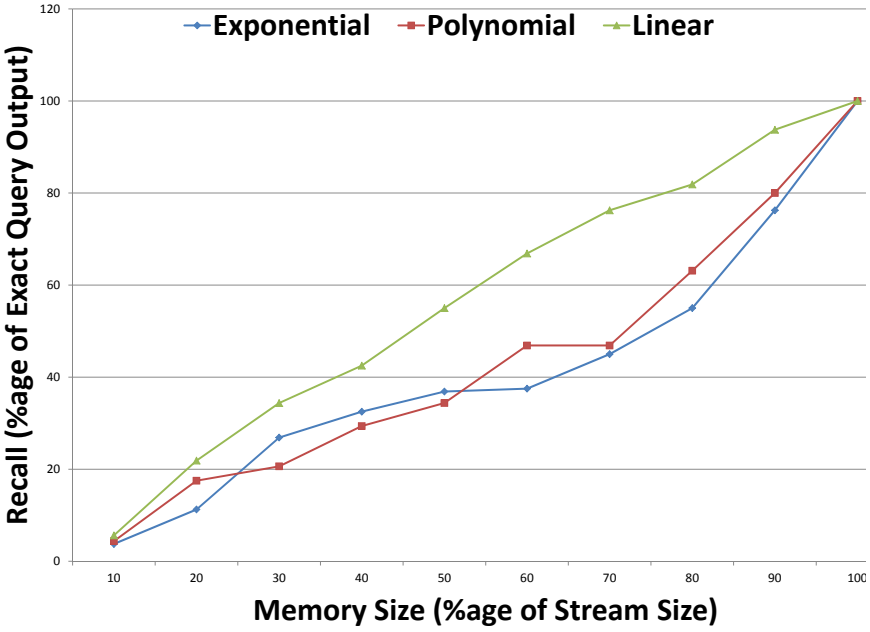


Figure 5.4. The Effect of Cost Function on Productivity for Real-Life Data (DBLP & Railroad).

joins between multiple streams. We could have compared our experimental result with a simple system that implements a fixed element-based or time-based sliding window. But intuitively, it would have been similar to a simple shedding implementation of first-in/first-out without any relevance to element's probable effectiveness in join state and would have resulted in less productivity. We have excluded several other types of joining queries, such as predicate-based joins, aggregation queries, and quantiles. The predicate-based join queries can be best addressed through combination of our semantic synopsis construction model in the previous chapter and the relevance synopsis of this chapter. Also, we have not studied the effect of time on utility-based relevance factor that is included in all of our cost functions. Similar to age-based

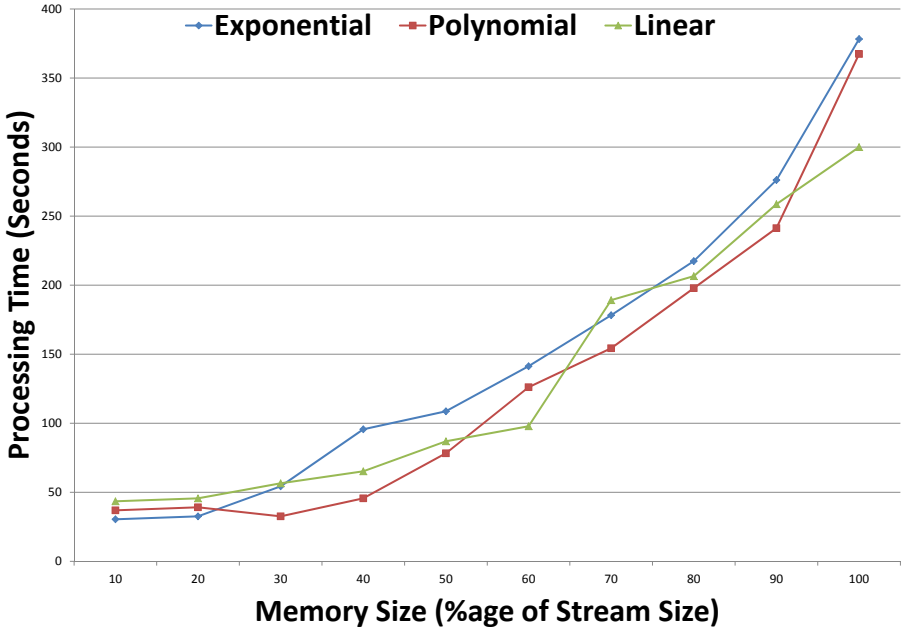


Figure 5.5. The Effect of Cost Function on Processing Time for Real-Life Data (DBLP & Railroad).

relevance factor the costing of this part could have been changed based on its time stamp of participation in the result.

CHAPTER 6

LOAD SHEDDING IN XML STREAM AGGREGATIONS

In this chapter, we extend the concept of relevance-based tuple expiration (proposed in the last chapter for join queries) for aggregation queries. However, in this chapter, we refine our relevance method a little to suite the requirements for aggregation. To prove the effectiveness of our approach, we test it in a pertinent field of e-commerce for a special case of aggregation in top-k problem, which also requires a join besides the aggregation.

6.1 Overview

Stream-based commercial applications, such as eBay, require to identify best-seller items on the fly 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 without an unbounded amount of memory. The problem becomes even harder when data in another source (stream) decides which item qualifies as more frequent than others, as it requires join between the two streams. Another real life example is top outlier detection of hot journals in a train fleet from hot box detector sensor messages, as described in Section 1.2. The data from numerous hot box detector sensors is to be joined with data from Automatic Equipment Identifier (AEI) sensors and aggregated to raise suitable alarm to

set out the culprit car. We have also tested our framework against this real world situation using the real world data as covered in the experimental section in this chapter.

The hot items or top-k ranking item queries are treated as iceberg queries [105] and solved using sliding window or sampling [106, 107, 108, 109, 64, 17, 67]. Use of synopsis or some form of statistics to answer the query is described in [110, 111]. Approximate algorithms, such as Counting, Hashing, Sampling and Sketches and Wavelet transforms [67], are used to solve this problem. The FREQUENT algorithm has been proposed in [112] that identifies the frequently occurring items in sliding windows and estimates their frequencies using a synopsis. They use both window-based synopsis and a global synopsis to determine the top-k items. A complete survey of top-k query processing techniques has also been covered for both relational and semi-structured databases in [113]. Frequent item calculation in a stream context has been proposed in [110, 114], which is closer to top-k and ranking algorithms. A solution through load shedding in relational stream joins has been addressed by [65, 64] through a randomized sketch in [66]. The issue of grouping is very challenging in the XML domain as they lack native grouping constructs in XML query languages [115]. Mining only top-k items from a single stream instead of monitoring all most frequent items has been covered in [108], whereas the problem of finding most popular k elements and finding frequent elements from a stream is covered in [109]. For indexing and searching schema-less XML documents based on concise summaries of their structural and textual content, Weimin He et al. presented two data synopsis structures in [116] that correlate the textual with the positional information in an XML document and improve the query precision.

We propose a solution that is constant in memory and provides acceptable quality in the result. Like previous chapters, we use a synopsis (a stream cube synopsis)

that takes into account the data distribution and cardinality of attributes and is very light-weight to create and maintain. It addresses the state management between two streams that determines the top-k list dynamically. It builds over the notion of data distribution of joining attributes. This provides two-fold benefits. (1) saves space by using a compressed synopsis that is built using domain compression technique, and (2) sheds unimportant tuples early on that reduces load on CPU. We have used the existing XMark auction data [103] and Hot Box Detector data (Appendix A.3) to show its effectiveness.

6.2 Aggregation Query Processing Model

In case of auction and bid stream [103], the auctions from an auction stream have to be stored for incoming bids in bid stream, which might come at a later time. This necessitates storing items from the auction stream beyond the window where they appear to have a possible join in the 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 results given the available memory. Besides, group-by queries pose a new challenge of the possible number of group-bys. With a n number of attributes $\{A_1, A_2, ..., A_n\}$ or dimensions having domain $\{d_1, d_2, ..., d_n\}$, there can be 2^n number of possible group-bys that can appear in aggregation queries [76]. This adds more overhead to the already precarious space requirement.

Our main objective is to determine the hot items in auction stream based on the count of bids in bid stream with a limited amount of memory and with the least possible error. To do this, one has to aggregate the bid stream on a window that spans from the open auction to the closed auction of the corresponding item. To determine comparative counts of bids for items to determine top-k hot items, these

windows must span over the life span of all auction items. To limit space, sometimes the problem is modified to 'what are the top-k hottest items in last one minute or so' [117]. Even for such a small time window, the available memory may not be adequate enough to capture counts of all bids for all possible n items when n becomes very large. A naive counting method for answering such queries, by maintaining a counter for each item, needs at least a memory of size of $\Omega(n)$ space complexity [112]. This necessitates a different approach that is put forth in this chapter, which reduces the number of counters by reducing the domain size of the auction items but effectively preserving the counts of individual data items.

For the sake of simplicity, we have limited our discussion to two streams $i = \{1, 2\}$, S_1, the auction stream and S_2, the bid stream. The auction items coming in S_1 are considered hot, based on the number of bids in S_2. As a result, the streams S_1 and S_2 need to be joined. In our approach, the size of the window $S_i[W_i]$ is determined on the basis of the available memory. Unlike exact analysis, where the window should envelop all possible time spans, our window is much smaller, having size proportional to 'k' based on the top-k value. We have pushed the aggregation (group bys) operation above the join, as shown in Figure 6.1, to avoid any blocking operation before possible joins. The aggregation operation is carried out to determine top-k in each category using the aggregation operator.

6.2.1 Compressed OLAP Data Cubes

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 [87]. This technique is only valid for continuous dimensions, such as age and salary, but

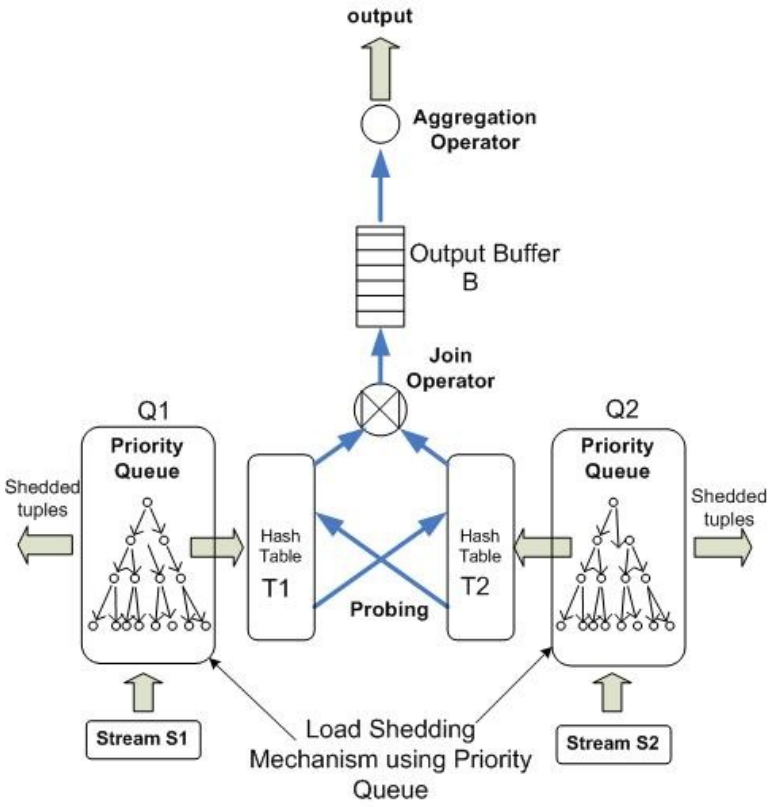


Figure 6.1. Overall Aggregation Query Processing Architecture.

dimensions, such as item category, are discrete and will not follow the pattern. Various compression techniques have been discussed in [118].

As we will show in Section 6.4.1, we use a hash function ($mod\ x$) to help us in compressing the cardinality of any attribute, such as 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 to be 100 for both streams 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 [87] to formulate this compression technique to reduce the space requirement for maintenance of bid counts. We have used a cost function, as shown in Equation 6.1 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 helps drive our load shedding process before the element is ch as an useful element and then is inserted into the hash buffer, in order to process the symmetric hash join with the other buffer.

6.3 The Synopsis for Aggregation

Our synopsis makes best utilization of limited computational resources (memory space and CPU) while answering aggregate queries over infinite streams. As our aim is to get the *top-k* hot items being auctioned, we can get a data distribution as shown in Figure 6.2, 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, *Pr(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 the *top-k* selling categories. Similarly, the same process can be applied to *top-k* items in each category. We transformed this continuous model into a discretized model to keep the counts of items (bids) as described in Section 6.3.1 that not only saves us buffer space but also helps us in calculating the probability of shedding for each tuple faster as described in Section 6.4.2. The heart of the synopsis is the construction and maintenance of this histogram like data structure on compressed dimensions.

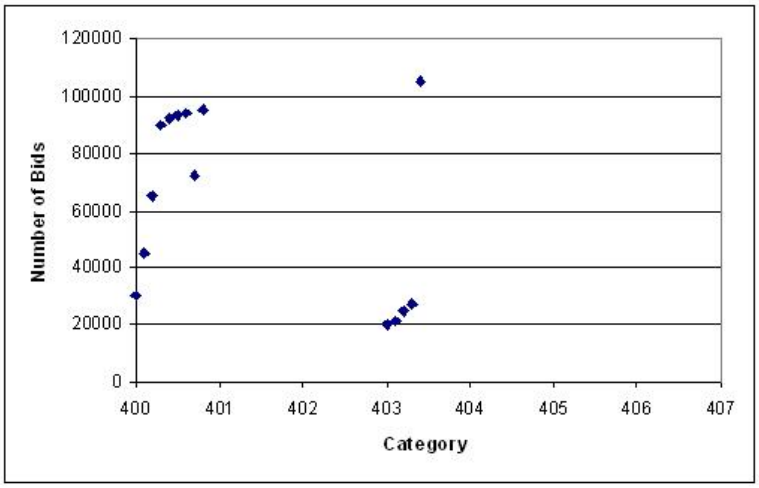


Figure 6.2. Sample Data Distribution.

6.3.1 Domain Compression

The use of stream cubes for streams is a topic well covered in [80] and to answer aggregate queries has been discussed in [76]. Various domain compression techniques are followed in the field of OLAP to save space for data cubes [87, 118]. The compression is achieved by exploiting the statistical structure of the data and its probability distribution along the domain and works well when the density distribution is continuous such as, salary and age. We have adopted a similar approach to keep statistics of our incoming tuples in a small space. But in our case, when the attribute *(itemid)* is not continuous, we follow the discretized model of domain compression and use the histogram model instead (Figure 6.3). Let the stream S_i have domain D_k for its attribute A_k. We have used a hash function as described in Section 6.2.1 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 the corresponding itemid flow in. Based on the value of these counters the probability of shedding is decided as in Section 6.4.2.

The domain compression not only saves us space for the synopsis construction and maintenance but also provides faster cost evaluation for shedding.

6.3.2 Cost Function

The calculation of weight for each element in the logical queue (priority queue) is done using the following cost function in Equation 6.1. Keeping the *top-K* result of the query in view, we have decided to shed non-frequent items before they consume critical computational resources. We have calculated probability of shedding for each incoming bid in 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 S_i. 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 6.2. Depending on the attribute value for this dimension, E_i 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 E_i is as follows.

$$P_{Ei} = 1 - p(\frac{C_{kj}}{\sum_{j=1}^{M} C_{kj}}) \tag{6.1}$$

6.4 The Load Shedding Mechanism for Aggregation

Besides the steps to reduce memory footprint, as described in Section 6.3, we have taken steps to reduce the load on CPU to speed up the processing to match up with the incoming data rate. As aggregation is a blocking operation, we have moved it after the join, as shown in Figure 6.1 to provide the join results early on. The load shedding operators are added to each incoming stream before the join takes place. This way, the un-important tuples go out of the queue 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. The load shedding is implemented through the priority queues as shown in Figure 6.1. This queue decides which tuple (node) to shed based on the weight of the node. The fast access to the priority queue helps faster decision on choosing the ideal victim tuples to shed, thus acting as a very fast shedding operator. We have built this framework on the top of our established system [9] and kept the process of join operation similar to it, but incorporated a different cost function based on probability rather than relevance, as described in Section 6.4.2.

We have adopted an approach similar to windowed join to process the join query. However, the window in our case is a fixed-size hash buffer for each source stream, as shown in Figure 6.1. 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. Upon arrival, the cost of an existing element in the heap is updated if this element matches the new incoming element; otherwise, the new element is added to the heap. If the action is to update, the heap is re-sorted through the heapify operation. On insert, the shedding decision is made if the buffer is already full. If the buffer is not full, the element is added to the heap and heapified and its reference is added to hash buffer. The shedding is decided if the cost of the new element is less than the cost of the element at the top of heap. Otherwise, the new element is dropped without any shedding. The shedding enacts the deleting of the element at the top and adding of the new element to the heap and removing its reference from the hash buffer. Once again, the heap gets heapified after the operation completes. The output tuples are queued into a buffer and then have to pass through an aggregation operator to calculate the top-k values. The aggregation operator is implemented to calculate both top selling

Table 6.1. Execution Steps (Refer to the Figure 6.1 for the notations used)

1. A new element e arrives in any stream S_2 (the bid stream)
Phase I: Shedding and updating the data structure Phase
2. If the element is already in hash table T_2
a. Update the element cost in the priority queue Q_2.
Else
b. Calculate the cost (C_e) of the element e, and try insert it into Q_2 (as in step 3).
3. If $C_e >$ cost of element at head of Q_2,
Throw element e without insertion
Else
Shed head of Q_2 and Insert element e (cost C_e) into Q_2 and correspondingly the reference in T_2.
Phase II: Join Phase
4. Probe T_1 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 S_1 symmetrically with converse data structures.

categories and top selling items in each category before sending out the result for any time span. The steps of processing is described in Table 6.1.

6.5 Experimental Evaluation

In this section, we present the results that we get from our system, implemented based on the framework that we presented in this chapter. We compare the results with our earlier implementation [9] that uses the relevance index as shedding mechanism and with exact result using Stylus Studio [101]. Our comparison is mostly based

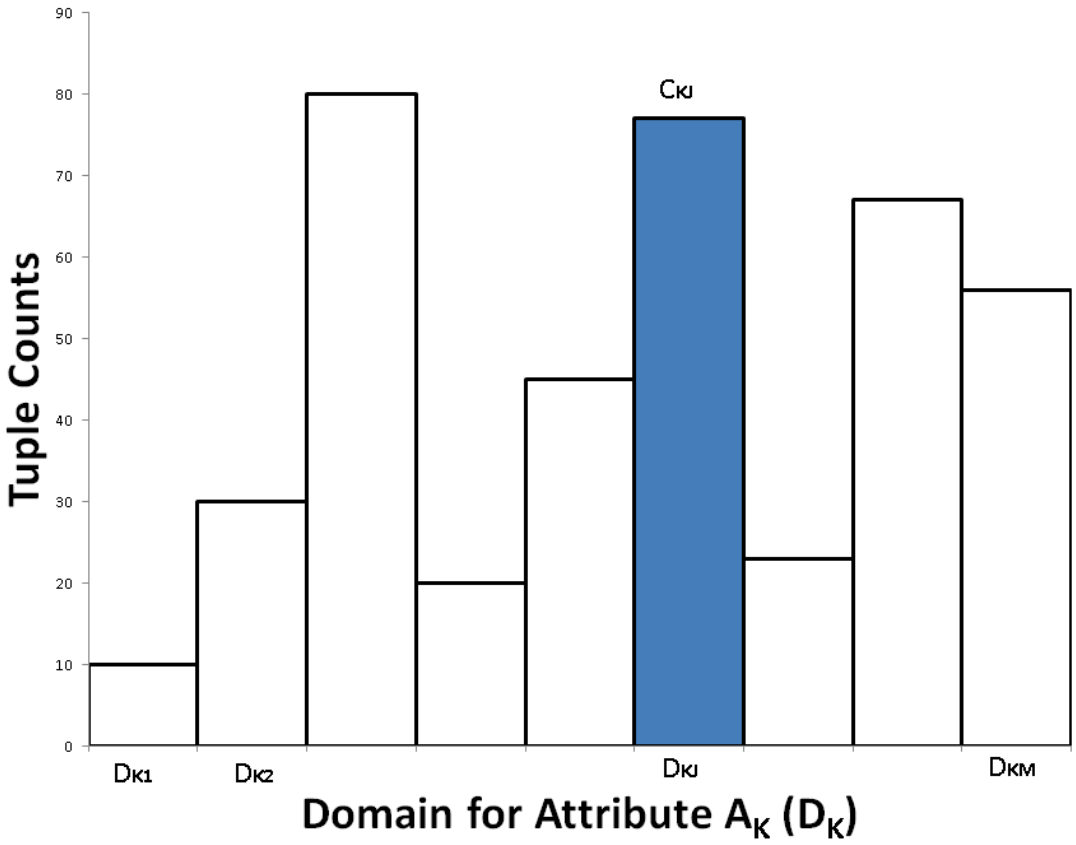


Figure 6.3. The Compressed Domain with Discrete Sub-Domains.

on three factors; the overall quality of the result, the overall memory consumption by system, and the processing time.

Our implementation is in Java, ran on a 2-GHz Intel Core 2 Duo machine with 2.0 GB of main memory running Windows XP. 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 have tested our framework on representative datasets derived from the auction data of XMark. The use of XMark auction data is to test the motivational scenario that is described in Section 6.1. The proposed framework could well be

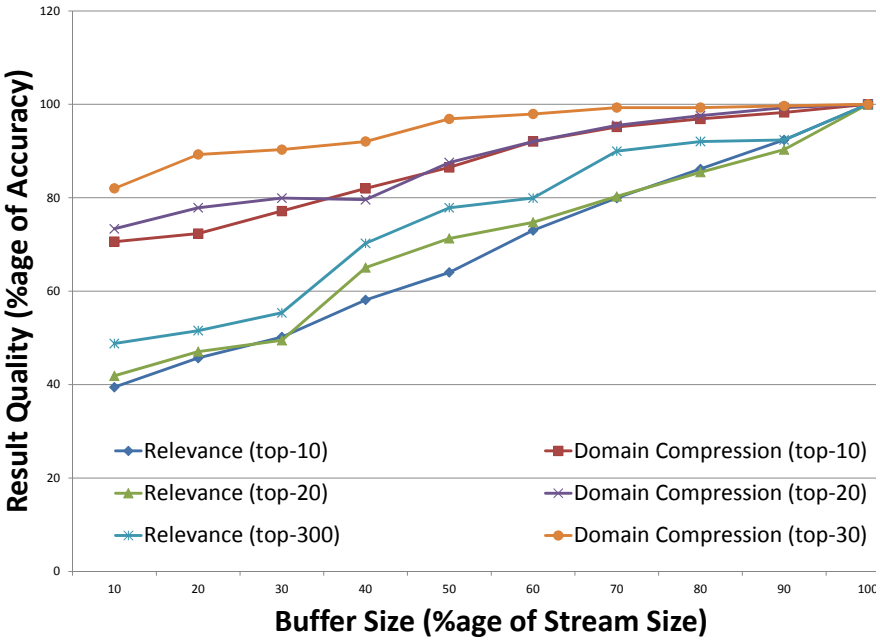


Figure 6.4. The Effect of Domain Compression and Relevance Shedding on result quality.

applied to the relational streams as well. The size of the dataset is controlled to avoid the memory limitations of the systems used [107]. 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 limitations for Stylus Studio. We ran five different set of aggregation queries, similar to Table 6.2 for different top-K values on all three implementations (our present scheme, the implementation of relevance index [9], and Stylus Studio). The queries have been modified to suit the implementation [111] to calculate both the hot list in each category and the hot categories. Instead of calculating the hot list over a time span, as in [117], we ran it for the entire length of stream. The definition of a hot list, as assumed by [117],

Table 6.2. Equivalent Relational Query

```
SELECT bid.itemid
  FROM bid [RANGE 60 MINUTES PRECEDING]
    WHERE (SELECT COUNT(bid.itemid)
       FROM bid [PARTITION BY bid.itemid
  RANGE 60 MINUTES PRECEDING])
     >= ALL (SELECT COUNT(bid.itemid)
     FROM bid [PARTITION BY bid.itemid
RANGE 60 MINUTES PRECEDING]);
```

is based on the number of bids an item receives rather than number of items being actually sold. The equivalent relational query provided by [117] is given in Table 6.2.

6.5.1 Shedding vs. Compression

Our first set of experiment is to see the effect of domain compression compared with the relevance index [9] 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; in our case of 50 MB for each stream. The other memory sizes are reduced according to the ratio. Figure 6.4 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 6.5 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.

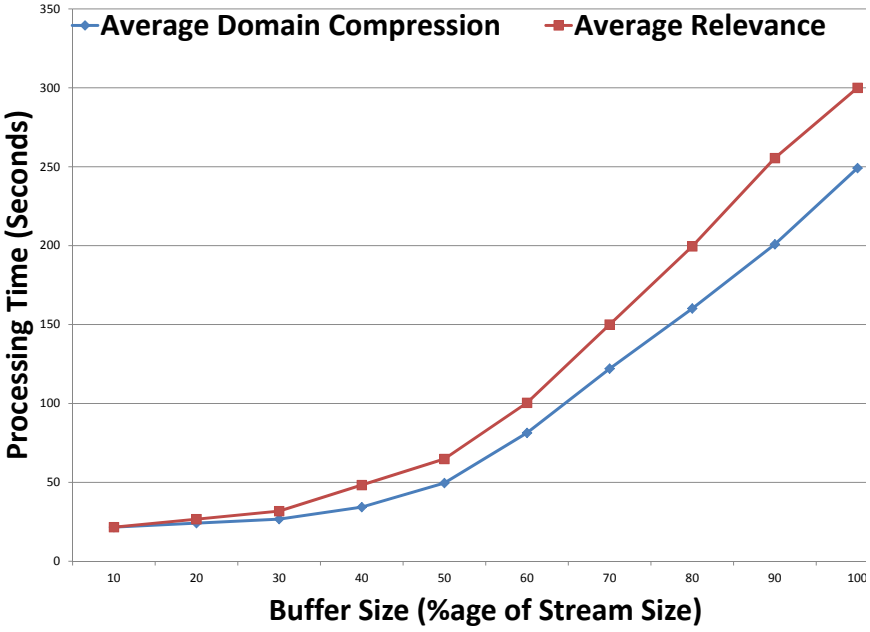


Figure 6.5. The Effect of Domain Compression and Relevance Shedding on Processing Time.

6.6 Summary

In this chapter, we presented a framework for load shedding for complex queries that yields approximate but high quality results. The heart of the framework is the construction of an intelligent synopsis that is light-weight both in space and time and helps processing aggregation queries with higher result quality in the face of limited resources and query complexity, such as joining. We applied this framework to the well-known auction benchmark to prove the effectiveness of our approach. Though domain compression is quite popular in OLAP queries, it is hardly being used streams. We have attempted to apply this concept into our synopsis construction and proved its effectiveness in load shedding through experimental results. The efficiency of the

compression is evident in the probability of shedding that results in higher quality of top-k query result.

Instead of 1-dimensional histogram, we could have used a multidimensional histogram that reflects other dimensions of the data, such as item category. This would help in answering aggregation queries on other dimensions as well. Also the access to statistics from different attribute perspective would have been easy and complete. We will extend this concept of dimension compression to joining scenario where the join will be carried over the cubes instead of the streams themselves in the next chapter.

CHAPTER 7

STREAM AGGREGATION AND JOIN USING STREAM CUBE

In this chapter, we focus on a special case of aggregation queries, namely, *group-by* queries. Similar to the previous chapter, we combine it with joins. Group-by aggregation queries are very common in *OLAP* and data streams. We solve this problem, with the help of a common OLAP tool, *Cubing*, to produce exact result. Our method, not only uses less resource (memory, execution time) but also produces *exact* query results. We explain this hybrid method through various algorithms and prove its effectiveness to solve this challenge through our cost models and prototype.

7.1 Overview

Multi-dimensional data stream processing shows characteristics from both analytical (OLAP) and transactional (OLTP) domains. They are analytical and monitoring in nature. The continuous processing of *Aggregation Join* queries over data streams shows the OLAP (group-by (aggregation) queries) and OLTP (join queries) characteristics. Common solution framework for this type of query is *windowing*, which produces partial results. The aggregation join query is the most complex query type as data streams are inherently multi-level and multidimensional in nature [84]. Given their large dimensionality, group-by/aggregations become more complex as the number of possible group-bys can be in the order of 2^n. For exact query answer, the group-bys and joins require unbounded memory as the size of the data is unbounded and these operations are blocking in nature. This complexity due to

multi-dimensionality is known as the '*curse of dimensionality*' [84, 85] in stream analysis.

We introduce a novel data structure (*Aggregation Cube*) to evaluate queries with multiple group-bys (aggregations) and joins. This data structure replaces the classical sliding window in the solution framework. Instead of performing a symmetric join between two streams and then construct a data cube, we perform aggregation for each stream through a data cube and then do the join, by joining the two cubes using a '*cube*-to-*cube* join'. The entire process is called a aggregated cube join *(ACJ)*. Our objective is to produce *exact query results*, rather than approximate results in sliding windows. Since data cube construction is a blocking operation, we extend our framework to cope with unbounded streams but producing continuous results as updates to the cube.

7.2 The Multidimensional Data Stream Model

We aim to reduce group-by aggregation queries to the problem of aggregation on stream cubes. In a N dimensional data stream there can be 2^N possible group-bys. The space requirement to answer all these group-bys is exponential. But the actual number of *group-bys* that a stream system typically needs is much smaller. The stream cube organizes the tuples as they come by into its proper space-based on its dimensional attributes. In *aggregate join queries*, the aggregation parameters, such as the group-by attributes, are known beforehand. Similar to [76], we limit the materialization of a stream cube to dependent cells only. The on-line computation of dependent cells is sufficient to answer all possible group-bys. The materialization is done for the corresponding group-bys. Keeping the small size of dependent cells in memory, we can produce a more accurate answer to group-by (aggregation) join queries.

7.2.1 Problem Definition

We model the data stream S as an infinite sequence of tuples T, represented as a set $\{T_1, T_2, ...\}$. Considering this stream as a N dimensional stream, each of the tuples T_i has N number of dimensional attributes and a measure M.

Definition 7.2.1. ***Tuple***. *A tuple $T_i = (A_1, A_2, ..., A_N, M)$. where A_1 through A_N are attributes for each of N dimensions and M is the measure. In the aggregation join context with possible group-bys, it can be also represented as $\{\{G, J, A\}, M_G\}$ where $\{G\}$ = set of grouping attributes, $\{J\}$ = set of joining attributes, $\{J\} \subset \{G\}$, $\{A\}$ = set of aggregating attributes, $\{M_G\}$ = set of measure that gets aggregated-based on the group-bys.*

Definition 7.2.2. ***Aggregation Cube***. *The aggregation of a stream can be represented as an Aggregation Cube. The Aggregation Cube for stream S, AC(S), is a set of aggregated cells. Each aggregated cell is a set of grouping attributes (G), aggregating attributes (A) that is represented as *, aggregating measures (M_G) on which the aggregation function F is applied to calculate the aggregated value $F(M_G)$ and count of tuples in the aggregation (K_G).*

$$AC(S) = \{G, A, F(M_G), K_G\}$$

The aggregation function F can be any of the functions, such as COUNT, SUM, MIN, MAX, and AVG.

Example 7.2.3. *Let stream S be a three-dimensional stream having tuples $T_i = (A_1, A_2, A_3, M)$. If we have a query that does the group-by by A_1 and A_2, then the grouping attributes are $G = \{A1, A2\}$, and the aggregating attributes $A = \{A3\}$, generally represented as *. The aggregation cube AC(S) for stream S can be represented as*

$$AC(S) = \{A_1, A_2, *, F(M_{A1A2}), K_{A1A2}\}$$

Definition 7.2.4. ***Aggregation Cube Update***. *The update process of the aggregation cube* $AC(S) = \{G, A, F(M_G), K_G\}$ *on the arrival of a new tuple* T_i *is as follows.*

1. T_i *updates the existing aggregated tuple or aggregated cell* $l = (G, A, F(M_G), K_G)$ *iff* $T_i.G = l.G$
2. *Otherwise, it adds a new aggregated tuple or aggregated cell* $l\,(G, A, F(M_G), K_G)$ *to aggregation cube AC(S) with its measure as the new aggregated tuple's measure.*

This way, the *Aggregation Cube Update*, controls the space requirement of the aggregation cube. The necessary aggregation tuple(s) l gets created, if it is necessary.

Definition 7.2.5. ***One-way Aggregation Cube Join***. *Consider running a group-by aggregation join query on two streams* S_1 *and* S_2 *having different number of attributes. Let the stream* S_1 *only contains the group-by attributes* $\{G\}$ *and the measure value* M *and has to be aggregated and joined with* S_2. *Let* (A_1, A_2) *are grouping attributes of* S_1 *and let both streams have at least one joining attribute* $\{J\}$, *which is part of the grouping attributes* $\{G\}$. *Then*

$S_1 = \{\{G, J, A\}, M_G\} = \{A_1, A_2, A_3, M\}$, $S_2 = \{J, A\} = \{B_1, B_2, B_3\}$,

such that $S_1.A_2 = S_2.B_2$.

Then, using Definition 7.2.2, the aggregation cube for stream S_1 *can be defined as,*

$$AC(S_1) = \{A_1, A_2, *, F(M_{A1A2}), K_{A1A2}\}.$$

The one-way aggregation cube join between $AC(S_1)$ *and* S_2 *can be represented as*

$$ACJ(S_1, S_2) = \{A_1, A_2, *, B_1, B_3, F(M_{A1A2}), K_{A1A2}\},$$

where $F(M_{A1A2})$ *is the aggregation measure and* K_{A1A2} *is the cardinality of the aggregation.*

Example 7.2.6. *Considering the one-way aggregation cube join between auction stream* S_1 *and bid streams* S_2 *for count of bids for each auction, our model is shown*

in Figure 7.1.

*$ACJ(S_1, S_2)$ is a join between aggregation cube $AC(S_1)$ and the bid stream S_2 and represented as $AC(S_1) \triangleright\triangleleft_{S1.auctionID=S2.auctionID} S_2$. Where $AC(S_1)$ is a set of aggregation tuples, $\{A_1, A_2, *, F(M_{A_1A_2}), K_{A_1A_2}\}$. The grouping attribute is auctionID, and other attributes is aggregating attributes (represented as *) and aggregating function is Count. Thus*

$$AC(S_1) \equiv \{auctionID, seller, startPrice, .., Count(auctionID), Count(auctionID)\}$$
$$\equiv \{auctionID, *, *, ..Count(auctionID), Count(auctionID)\}$$

The one-way join between $AC(S_1)$ and stream S_2 produces a continuous sequence of tuples

$$ACJ_{out} = \{S_1.auctionID, Count(S_2.auctionID)\}$$

7.3 The Aggregation Join Query Processing Model

In this section, we extend the concept of aggregation cube join to two streams where both have aggregation attributes. We formalize this case both for late aggregation and early aggregation using aggregation cube join method in the next two subsections.

7.3.1 The Late Aggregation Model

This model implements aggregation processing after the join. As shown in Figure 7.2, the streams S_1 and S_2 have joined over their joining attributes $\{J_1\}$ and

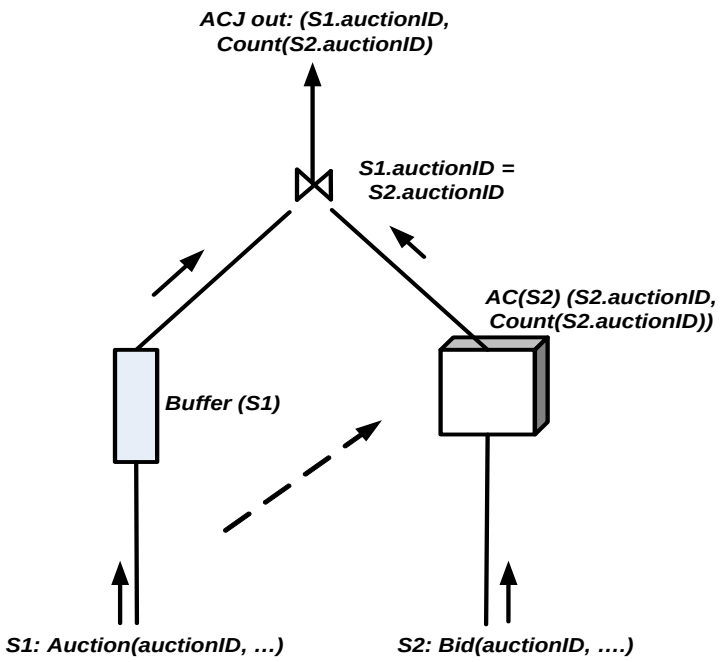


Figure 7.1. An Example of One-Way Aggregation Cube Join.

$\{J_2\}$. The aggregation is processed over the result of the join to produce final output. Once again, the aggregation is processed through an aggregation cube. The Streams

$$S_1 = \{A_1, A_2, A_3, M_A\}$$

$$and \quad S_2 = \{B_1, B_2, B_3, M_B\},$$

where A_1, A_2, A_3 are dimensional attributes of stream S_1 and B_1, B_2 and B_3 are dimensional attributes of stream S_2. M_A and M_B are measures of tuples for stream S_1 and stream S_2 respectively.

Let the aggregation join query be the following query, applied to these two streams. Then A_2 and B_2 are the join attributes of S_1 and S_2 such that $S_1.A_2 = S_2.B_2$.

```
SELECT *, SUM(MA, MB)
```

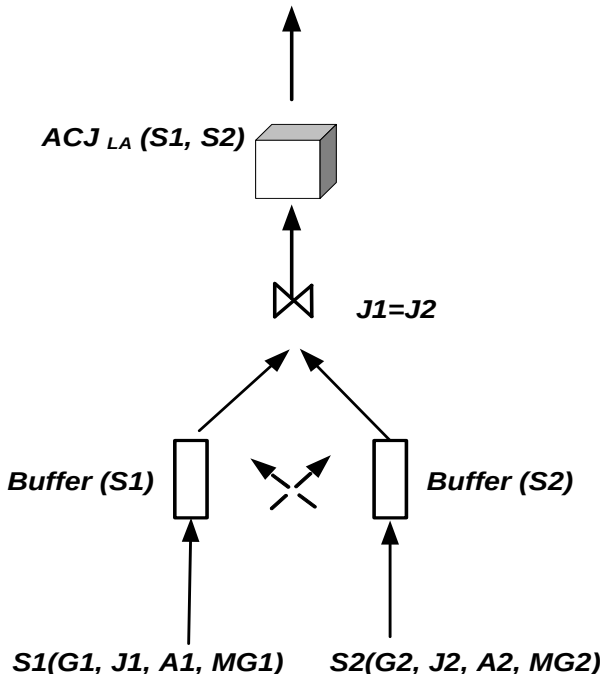


Figure 7.2. Late Aggregation in Two-Way Aggregation Cube Join.

```
FROM Stream S1, Stream S2
WHERE S1.A1 = S2.B2
GROUP BY S1.A1, S1.A2, S2.B2, S2.B3
```

Step I (Stream Join) - The first step, which is a join between S_1 and S_2 as shown in Figure 7.2, will produce a set of tuples that feeds the second step of aggregation. The output from first step can be represented as

$$\{A_1, (A_2 = B_2), A_3, B_1, B_3, (M_A + M_B)\}$$

Step 2 (Aggregation) - With A_1, A_2, B_2, B_3 as group-by attributes, the result from the second step can be represented as

$$ACJ_{LA}(S_1, S_2) = \{A_1, (A_2 = B_2), *, *, B_3, F(M_A + M_B)\}$$

7.3.2 The Early Aggregation Model

In this model, we do the aggregation phase first and the join thereafter. This is our preferred model, in which the group-by aggregations will be carried out through aggregation cubes and the join will be between these aggregation cubes. Once again, we use the same two streams as described in the previous subsection.

Step I (Aggregation Step) - If we apply the aggregation cube as defined in Definition 7.2.2 to both streams, the resulting aggregation cubes for respective streams can be represented as

$$AC(S_1) = \{A_1, A_2, *, F(M_A), K(A_1, A_2)\}$$

$$AC(S_2) = \{*, B_2, B_3, F(M_B), K(B_2, B_3)\}$$

Step 2 (Cube Join) - If we process the join between $AC(S_1)$ and $AC(S_2)$, such that $AC(S_1).A_1 = AC(S_2).B_2$ then the result stream will be a set of tuples represented as

$ACJ_{EA}(S_1, S_2) = \{A_1, (A_2 = B_2), *, *, B_3, K(B_2, B_3) * F(M_A) + K(A_1, A_2) * F(M_B)\}$

Theorem 7.3.1. *Let $ACJ_{LA}(S_1, S_2)$ be the set of result tuples produced from Section 7.4.1 due to late aggregation on stream joins on S_1 and S_2 and $ACJ_{EA}(S_1, S_2)$ be the result set of tuples from early aggregation as described in section 7.4.2.* ***Then, for every tuple in*** $ACJ_{LA}(S_1, S_2)$***, there is a tuple in*** $ACJ_{EA}(S_1, S_2)$ ***and vice versa.***

7.4 Algorithms and Cost Models

We have produced an algorithm that explains the processing steps for an aggregation-join query on two streams (S_i, S_k) following the model shown in Figure 7.3. This algorithm is general enough to extend to more than two streams.

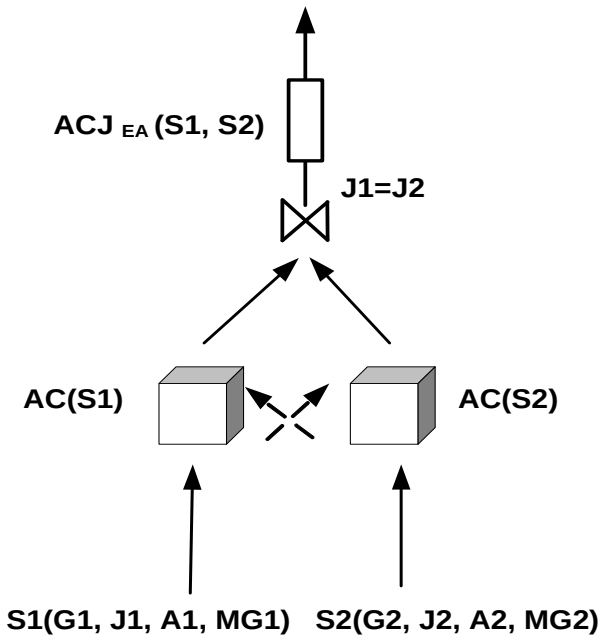


Figure 7.3. Early Aggregation in Two-Way Aggregation Cube Join.

As described in the next section, our implementation of aggregation cubes is based on fast search and fast insertion. The hashing of grouping attributes and hashing of joining attributes play an important key role in the execution plan of the prototype.

7.4.1 The Aggregation Cube Join Algorithm

Table 7.1 outlines the sequence of operations followed in the aggregation-cube join between S_i and S_k. Our assumptions are that the grouping attributes, aggregating attributes, and joining attributes are known apriori for each stream from the analysis of the standing query.

Table 7.1. The Aggregation Cube Join Algorithm

Phase I: Aggregation

INPUT: Streams (S_i, S_k), Grouping and aggregating attributes: (for $S_i - \{G_i, A_i\}$), (for $S_k - \{G_k, A_k\}$), Aggregation function F

OUTPUT: Aggregation Cubes: $S_i - AC(S_i)$, $S_k - AC(S_k)$

1. foreach *tuple s arriving in stream* S_i **do**
2. Find the aggregating tuple $(G, A, F(M_G), K_G)$ in the cube $AC(S_i)$ that maps to the grouping attribute value of s.
3. **If** *there is one* **then**
4. Update the aggregating tuple's aggregation function value of the measure $F(M_{G_i})$ with s's measure and increment the counter K_{G_i}
5. **else**
6. Create a new aggregating tuple $(G, A, F(M_G), K_G)$ and insert it in $AC(S_i)$ and update its aggregation function value $F(M_{G_i})$ with s's measure and increment the counter K_{G_i}
7. Update H_{insert} and H_{join} (Refer Figure 7.4) for the grouping attributes G_i and joining attributes J_i of the new tuple.
8. **end**
9. end

(Follow the same steps for any tuple s arriving in stream S_k)

Phase II: Cube Join

INPUT: Aggregation Cubes: for S_i - $AC(S_i)$, for $S_k - AC(S_k)$, Suitable hashing mechanism for joining attributes

OUTPUT: Output set buffer (holds all output tuples and need to be constantly updated/added)

1. foreach *aggregation tuple update in* $AC(S_i)$ **do**
2. Probe $AC(S_k)$ by using the hash value of the joining attribute(s)
3. **If** *there is a match* **then**
4. Search for the output tuple in output set that matches the grouping attribute set of the join
5. **If** *there is a match* **then**
6. Increment the final measure of the output set tuple
7. **else**
8. Create a new output tuple and insert that into output set
9. **end**
10. **else**
11. no output
12. end

Follow the same steps for any tuple update in $AC(S_k)$

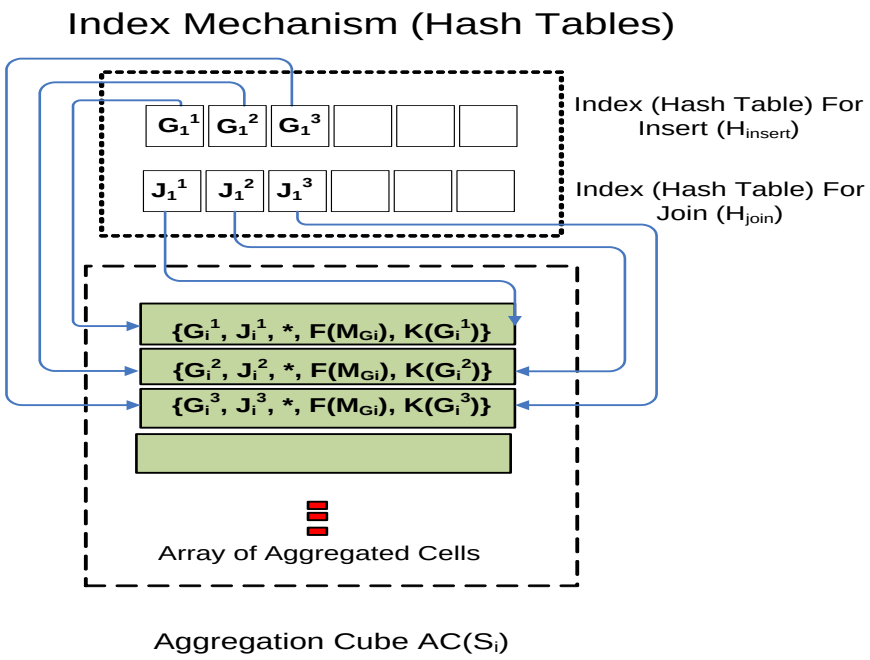


Figure 7.4. Implementation of the Aggregation Cube and Access Mechanism.

7.4.2 Cost Models

We have developed cost models for space and execution time for ACJ_{LA} and ACJ_{EA}, as shown in Figure 7.2 and Figure 7.3 respectively. We considered the two-way joins between two streams, where grouping attributes and measures are there in each stream. It requires construction of aggregation cubes for stream for ACJ_{EA} and construction of buffers to accommodate entire stream for each stream in ACJ_{LA}. We followed the concept of costing for time used in [73].

7.4.2.1 Cost Models for Time

Using the notation in *Table 7.2*, the cost for processing an aggregation join query in case of **Late aggregation**, as shown in Figure 7.2, is

Table 7.2. Cost Notations

Notation	Description
λ_{S_i}	Stream rate for stream S_i, $i = (1, 2)$
$C_{ACJ_{LA}}$	Cost of Aggregation Cube Join in case of late aggregation (Figure 2)
$C_{ACJ_{EA}}$	Cost of Aggregation Cube Join in case of early aggregation (Figure 3)
$C_{UB}(B_i)$	Cost of updating the buffer for each new tuple, $i = (1, 2)$
$C_{UC}(AC(S_i))$	Cost of updating the aggregation cubes, $i = (1, 2)$
C_{UO}	Cost of updating output set Buffer
$C_f(B_i)$	Cost of finding matching tuple in opposite bufferB_i, $i = (1, 2)$
$C_f(AC(S_i))$	Cost of finding matching aggregation tuple in opposite $AC(S_i)$, $i = (1, 2)$
m_i	Join cardinality of stream S_i, $i = (1, 2)$
n_i	Join cardinality of cube $AC(S_i)$, $i = (1, 2)$

$$C_{ACJ_{LA}} = \lambda_{S_1}[C_{UB}(B_1) + C_f(B_2) + m_2 C_{UC}(ACJ_{LA})]+$$

$$\lambda_{S_2}[C_{UB}(B_2) + C_f(B_1) + m_1 C_{UC}(ACJ_{LA})] \tag{7.1}$$

Similarly, the cost of execution of time for processing in case of **Early Aggregation**, as shown in Figure 7.3, is as follows.

$$C_{ACJ_{EA}} = \lambda_{S_1}[C_{UC}(AC(S_1)) + C_f(AC(S_2)) + n_2 C_{UO}]+$$

$$\lambda_{S_2}[C_{UC}(AC(S_2)) + C_f(AC(S_1)) + n_1 C_{UO}] \tag{7.2}$$

Our objective is to show that the aggregation cube join in general is better than window-based aggregation joins, particularly following early aggregation cube joins. Thus from above equations $C_{ACJ_{EA}} \ll C_{ACJ_{LA}}$.

Though we have showed it through our experiments, yet intuitively it can be formally proved as follows.

Comparing the various terms in Equations 7.1 and 7.2, the third term in Equation 7.2 is much less than that in Equation 7.1. As the output set buffer is much smaller compared to the result cube ACJ_{LA} in case of late aggregation, so also is the cost to update it. Thus, with confidence, we can say that

$$\lambda_{S_1}[m_2 C_{UC}(ACJ_{LA})] + \lambda_{S_2}[m_1 C_{UC}(ACJ_{LA})] \approx \lambda_{S_1}[n_2 C_{UO}] + \lambda_{S_2}[n_1 C_{UO}]$$

Considering the unbounded size of streams S_1 and S_2, the buffers B_1 and B_2 must be very large, which requires long updates. On the other hand, the aggregation cubes AC(S_1) and AC(S_2) have been implemented very efficiently for searching and inserting as described in Section 7.5.1. Thus, it can be intuitively said that the cost of finding a match in buffers in case of late aggregation will outweigh the cost of updating and joining in case of early aggregation, as shown in Equation 7.2.

$$\lambda_{S_1}[C_f(B_2)] + \lambda_{S_2}[C_f(B_1)] \quad \gg \quad \lambda_{S_1}[C_{UC}(AC(S_1)) + C_f(AC(S_2))] + \lambda_{S_2}[C_{UC}(AC(S_2)) + C_f(AC(S_1))]$$

Thus we can say that $C_{ACJ_{EA}} \ll C_{ACJ_{LA}}$.

7.4.2.2 Cost Model for Space

As our aim is to get exact answers to the posed aggregation join queries, unlike the sliding window methodology, we need to store the entire stream for S_1 and S_2 to get all possible joins between them.

The space required in case of ACJ_{LA}

$$S_{ACJ_{LA}} = |S_1| + |S_2| + |ACJ_{LA}| \tag{7.3}$$

Similarly the space required in case of early aggregation (ACJ_{EA}), as shown in Figure 7.3, is

$$S_{ACJ_{EA}} = |AC(S_1)| + |AC(S_2)| + |ACJ_{EA}| \tag{7.4}$$

Our objective is to show that the space required in case of early aggregation is better than that for late aggregation: $S_{ACJ_{EA}} \ll S_{ACJ_{LA}}$. From intuition, as we are storing only aggregated cells of the stream cube, the size of this array is of fixed size and does not change based on the size of the stream. Thus, it is **non-monotonic** with respect to the stream size. Rather, it depends on the number of grouping attributes and their domain size. Assuming that there is a limited number of standing queries in the system and they have overlapped grouping attributes, the number of independent grouping attributes will be fairly small [76, 78]. Thus, with fair amount of confidence, we can say that

$$|AC(S_1)| + |AC(S_2)| \ll |S_1| + |S_2|$$

And thus $S_{ACJ_{EA}} \ll S_{ACJ_{LA}}$.

7.5 Experimental Evaluation

The various parameters that affect the execution time and memory requirement for stream *aggregation join* processing are (1) number of grouping attributes, (2) stream rate, (3) size of the streams (in terms of number of tuples), (4) join selectivity factor and (5) number of join attributes. We have studied the effect of number of grouping attributes and stream size in detail through our experiment and the result is shown in Figures 7.5, 7.6, 7.7 and 7.8. We have kept the stream rate constant at 1000 milliseconds (average tuple arrival frequency) for all of our experiments. We have limited the number of join attributes to one for simplicity. The maximum number of dimensions has been limited to 8.

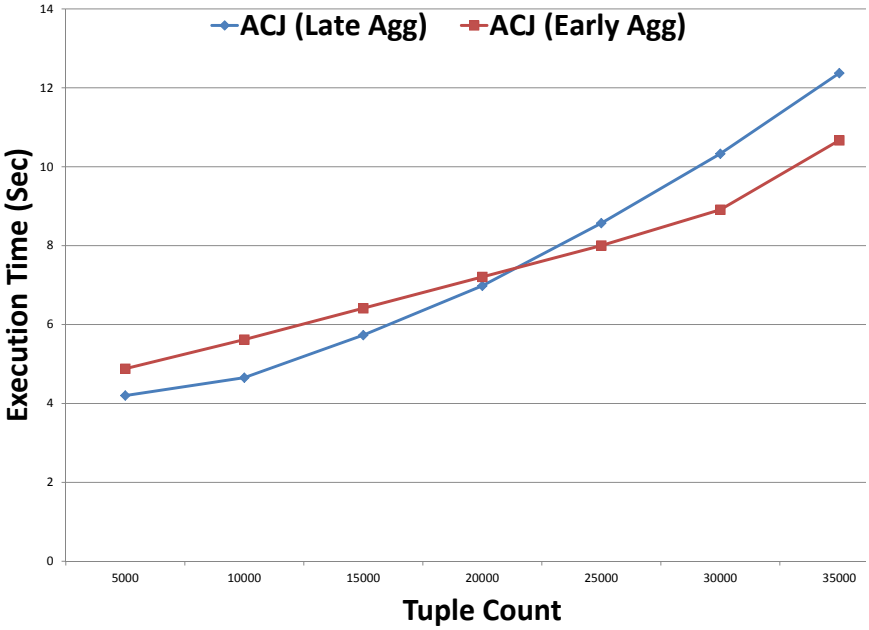


Figure 7.5. The Effect of Stream Size on Execution Time.

7.5.1 Implementation

We have implemented the aggregation cube as a list of aggregated cells [78] as shown in Figure 7.4. The aggregate cells are kept in a list and referenced through two hash tables for faster update and join. The hash table H_{insert} is keyed on the basis of grouping attributes (G_i) and the hash table H_{join} is hashed through joining attributes (j_i). Each time a new tuple arrives, new keys are to be inserted into H_{insert} and H_{join}, based on if a new aggregated cell is created in $AC(S_i)$ or not as described in Table 1. Both H_{insert} and H_{join} are very space efficient as they stores the reference to aggregated cells as values. This provides a highly efficient mechanism to carry out insert of new aggregated cells and join against the other cube.

7.5.2 Setup

We have implemented the aggregation cube join between two relational streams for both *late aggregation* and *early aggregation* (Figure 7.2, Figure 7.3). We have limited our implementation to two-way joins for simplicity. The program is in Java and runs on a windows XP with Core 2 Duo processors and 2.0 GB of RAM. Inputs are synthetically generated that takes number of tuples, number of dimensions, domain limits for each dimensions and stream rate as parameters to generate these data streams. The query is fed ahead of time for preprocessing to get grouping attributes, joining attributes and measure attributes etc. Value of each attribute is generated randomly with uniform distribution. We chose the data types for all attributes as integer data types, though they could be string data types for non-measure attributes. We control the number of distinct values for join attribute to control the join selectivity factor.

7.5.3 The Effect of Stream Size

Execution Time-We have used the number of tuples as an indirect measure of stream size. As shown in Figure 7.5, we pay little more price for early aggregation cube join up front, due to some pre-processing and setup costs for setting up aggregation cells, yet for higher stream size, it performs better than late aggregation cube join. As the size of the stream increases, so also the probing time for join processing in ACJ_{LA}.

Memory-We have measured the memory in terms of percentage of the maximum stream size (buffer size), which is 35000 tuples in our case. Figure 7.6 shows clearly the advantage of early aggregation cube join over the late aggregation cube join. The use of stream cubes in ACJ_{EA} has reduced the rate of usage of memory as the stream

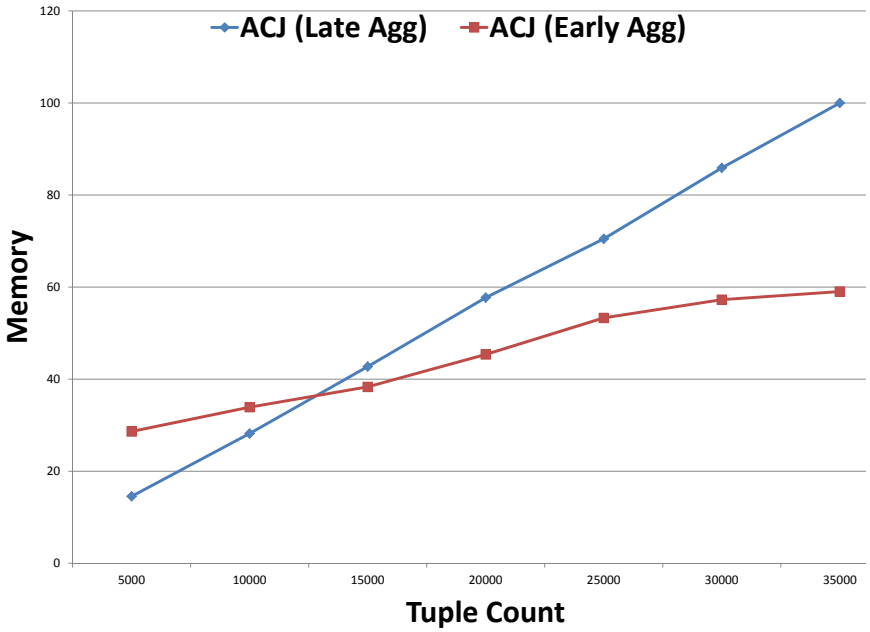


Figure 7.6. The Effect of Stream Size on Memory.

size increases. On the other hand, the ACJ_{LA} pays penalty of use of buffer which is directly proportional to the stream size.

7.5.4 The Effect of Grouping Attributes

Execution Time-Keeping the stream size fixed at 5000 tuples for each stream and varying the number of group-by attribute count, we measured the execution time for ACJ_{LA} and ACJ_{EA}. As shown in Figure 7.7, the ACJ_{EA} is a clear winner. The higher the number of grouping attributes, the better is the performance of ACJ_{EA}. **Memory**-From Figure 7.8, from the rate of change of memory requirements with respect to the grouping attribute count, it is evident that the number of grouping

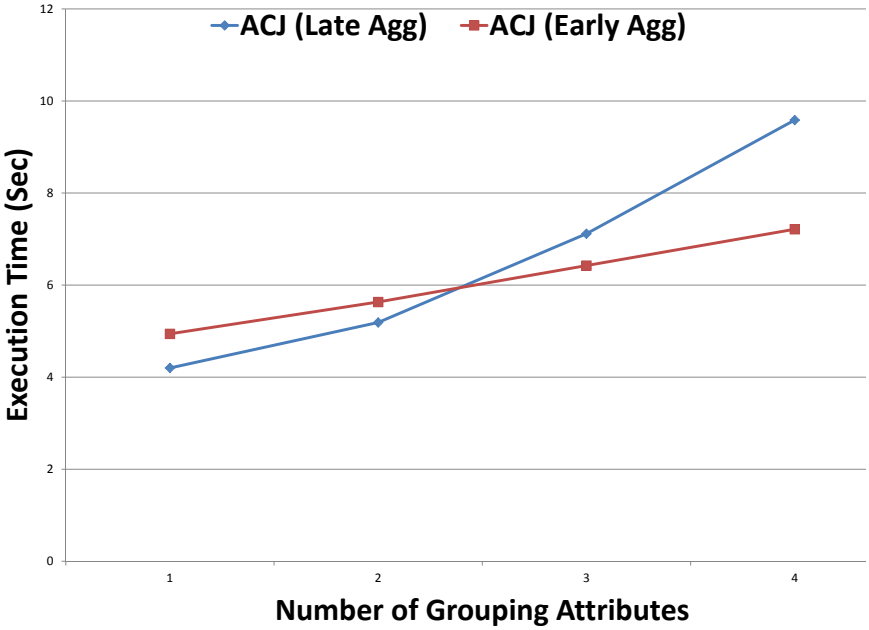


Figure 7.7. The Effect of Number of Grouping attributes on Execution Time.

attributes play a small role in differentiating the ACJ_{EA} from ACJ_{LA} compared to what the execution time does.

7.6 Summary

In this chapter, we addressed the issue of resource requirements for an OLAP-like situation where all possible group-by queries over data streams require unbounded resources for exact answer. We have proposed a new framework to get exact answers from these types of complex queries, applying the concept of stream cubes which is fixed in space. Through our models, experiments and algorithms, we have showed that this framework yields exact answers under manageable resources.

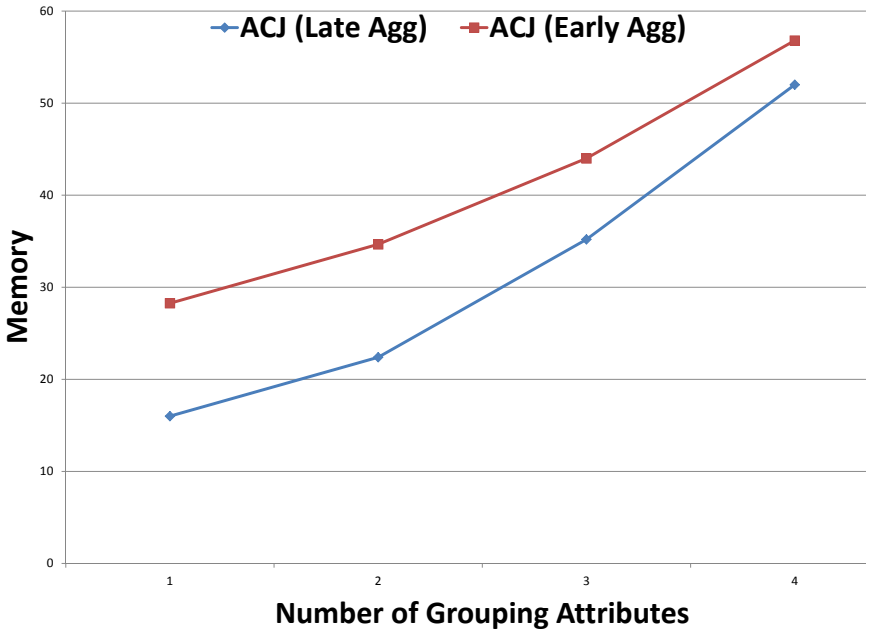


Figure 7.8. The Effect of Number of Grouping Attributes on Memory.

CHAPTER 8

CONCLUSION AND FUTURE WORK

During the course of this thesis, we presented a comprehensive framework for load shedding in various processing types over XML streams. The central philosophy of this framework is a load shedding strategy, that is the least intrusive from processing point of view and the least resource consuming. From a design perspective, our load shedder is developed as a separate module, external to the query operator network and hence can be pluggable to any stream processing engine. The very placement of our load shedder makes it proactive and compact from implementation standpoint. Our load shedding strategy is aimed at delivering the best QoS to the overall processing and is itself, based on a framework that is driven by the quality of data as input and output.

We have introduced a logical window model that is different from the established sliding window concept and spans over the entire history of the data stream; thus producing qualitative maximum size subset results. We used the concept of the relevance windowing for three types of queries (set-valued, aggregation, and join). Our synopses that drive the load shedding mechanisms, are resource-efficient and effective to construct and maintain.

We have planned to extend this work of load shedding in XML stream processing to the following areas.

Integration with static data - In this thesis, we primarily process XML stream data. However, most real world data stream systems need to integrate data between streaming and static databases. For example, the HBD messages (Appendix

A.3) are needed to be integrated with historical temperature statistics to transform the raw messages from sensors (A.3.1) to actionable data (A.3.3). We are planning to integrate the static data with streaming data in our system. As the requirements are different for streaming and static data processing, it will be challenging 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 be processed by influencing the selectivity of stream data.

Tuning of Relevance Algorithms - Our shedding algorithm for stream joins (Chapter 5) do not take into account the effect of age in output relevance. Though, we factor in the frequency (utility) in the input and output of any path element and age in the input stream in the cost calculation (Section 5.3.1), we have not taken into account the age of the resulted element in our shedding strategy.

Integration with CEP systems - Our stream processing system can play a role of a core query processing system, feeding results into the event notification layer of any Complex Event Processing System. Though load shedding is not prominent 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 drops or discards any events. Our time-based relevance model can play a crucial role in reducing temporal footprint of events in a event processing system. As the consumption modes are similar to logical window than a physical window (refer Section 5.2), our synopsis-driven relevance model is a good fit for them.

Stream Cube - We briefly introduced the stream cube as an effective synopsis in answering group-by queries in Chapter 7. We have implemented it for relational streams. However we are in the process of extending it to accommodate XML data streams.

APPENDIX A

REAL-LIFE SAMPLE XML STREAM DATA

In this appendix, we present various schema/DTDs of real life data and small excerpts of the data. We use this real-life data in our experimental evaluations along with synthetic XMark and DBLP data. This real-life data, is drawn from various sensors in Railroad field. Their specific usage is described in section 1.2.

A.1 CONSIST MESSAGE

The AEI (Automatic Equipment Identifier) senses the passing of a train over it. It sends the message to back office regarding details of equipment that it reads. This message contains details of car and locomotives in the train consist and the train. The schema of the consist message is given in Table A.1 and a sample XML consist message in Table A.2.

A.2 OS MESSAGE

OS messages are generated by signal control points as a train passes by it in a CTC (Computerized Track Control) territory. This message is the backbone of traditional train dispatching systems. The schema for this message is in Table A.3 and a sample data in Table A.4.

A.3 HOT BOX DETECTOR MESSAGE

The hot box detector is blind to train and car identifications. It only picks up the axle sequence that it is measuring against. The data further joined with AEI messages (A.1) and OS message (A.2) to enrich with car and locomotive information. After integration it results in a schema similar to Table A.5 and the sample data as in Table A.6. Before any actionable events gets created, this stream data gets correlated with historic HBD data to calculate k value and *zscore* value for the respective journal

Table A.1. AEI Consist Message Schema

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="ConsistMessage">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="CompanyAbbreviation" type="xs:string"/>
        <xs:element name="TrainSymbol" type="xs:string"/>
        <xs:element name="ReaderId" type="xs:string"/>
        <xs:element name="ReadingDate" type="xs:string"/>
        <xs:element name="ReadingTime" type="xs:string"/>
        <xs:element name="Station333" type="xs:string"/>
        <xs:element name="StationState" type="xs:string"/>
        <xs:element name="StationSeqNum" type="xs:string"/>
        <xs:element name="ConsistCarCount" type="xs:positiveInteger"/>
        <xs:element name="Equipment" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="EquipInitial" type="xs:string"/>
              <xs:element name="EquipNumber" type="xs:string"/>
              <xs:element name="LoadOrEmpty" type="xs:string"/>
              <xs:element name="TrainSeqNum" type="xs:positiveInteger"/>
              <xs:element name="AxleCount" type="xs:positiveInteger"/>
              <xs:element name="Length" type="xs:positiveInteger"/>
              <xs:element name="CarKind" type="xs:string"/>
              <xs:element name="AxleCountOrient" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

so that any trends can be verified and alerts can be raised. After joining with static historic data the raw message data looks like Table A.7.

Table A.2. Sample AEI Message

```
<ConsistMsg>
  <CompanyAbbreviation>BNSF</CompanyAbbreviation>
  <TrainSymbol>M BARRIC1 22A</TrainSymbol>
  <ReaderId>1234567</ReaderId>
  <ReadingDate>022412</ReadingDate>
  <ReadingTime>0821</ReadingTime>
  <Station333>PITTSBURG</Station333>
  <StationState>CA</StationState>
  <StationSeqNum>00970</StationSeqNum>
  <ConsistCarCount>115</ConsistCarCount>
  <Equipments>
    <Equipment>
      <EquipInitial>BNSF</EquipInitial>
      <EquipNumber>      7208</EquipNumber>
      <LoadOrEmpty> </LoadOrEmpty>
      <TrainSeqNum>1</TrainSeqNum>
      <AxleCount>6</AxleCount>
      <Length>73</Length>
      <CarKind>ENG </CarKind>
      <AxleCountOrient>A</AxleCountOrient>
    </Equipment>
    <Equipment>
      <EquipInitial>BNSF</EquipInitial>
      <EquipNumber>      3118</EquipNumber>
      <LoadOrEmpty> </LoadOrEmpty>
      <TrainSeqNum>2</TrainSeqNum>
      <AxleCount>4</AxleCount>
      <Length>59</Length>
      <CarKind>ENG </CarKind>
      <AxleCountOrient>B</AxleCountOrient>
    </Equipment>
    ...
  </Equipments>
</ConsistMsg>
```

Table A.3. OS Message Schema

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="OSMessage">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="LocationGuid" type="xs:positiveInteger"/>
        <xs:element name="LocationCadNumber" type="xs:positiveInteger"/>
        <xs:element name="ControlPointName" type="xs:string"/>
        <xs:element name="OsDirection" type="xs:string"/>
        <xs:element name="Station333" type="xs:string"/>
        <xs:element name="State" type="xs:string"/>
        <xs:element name="TrainnSheetDir" type="xs:string"/>
        <xs:element name="ArrivalTrackAsid" type="xs:positiveInteger"/>
        <xs:element name="DepartureTrackAsid" type="xs:positiveInteger"/>
        <xs:element name="ArrivalTrackGuid" type="xs:positiveInteger"/>
        <xs:element name="DepartureTrackGuid" type="xs:positiveInteger"/>
        <xs:element name="ArrivalTrackName" type="xs:string"/>
        <xs:element name="DepartureTrackName" type="xs:string"/>
        <xs:element name="TerritoryType" type="xs:string"/>
        <xs:element name="OsReportDateTime" type="xs:dateTime"/>
        <xs:element name="OsReportTimeZone" type="xs:string"/>
        <xs:element name="OsTimeUtc" type="xs:dateTime"/>
        <xs:element name="MilePost" type="xs:double"/>
        <xs:element name="MilePostModifier" type="xs:string"/>
        <xs:element name="TypeMove" type="xs:string"/>
        <xs:element name="LineSegment" type="xs:int"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Table A.4. Sample OS Message

```
<OSMessage>
  <LocationGuid>866007</LocationGuid>
  <LocationCadNumber>1753</LocationCadNumber>
  <ControlPointName>COOLIDGE</ControlPointName>
  <OsDirection>W</OsDirection>
  <Station333>COOLIDGE</Station333>
  <State>KS</State>
  <TrainnSheetDir>W</TrainnSheetDir>
  <ArrivalTrackAsid>2017</ArrivalTrackAsid>
  <DepartureTrackAsid>2017</DepartureTrackAsid>
  <ArrivalTrackGuid>866266</ArrivalTrackGuid>
  <DepartureTrackGuid>866272</DepartureTrackGuid>
  <ArrivalTrackName>M</ArrivalTrackName>
  <DepartureTrackName>M</DepartureTrackName>
  <TerritoryType>T</TerritoryType>
  <OsReportDateTime>2012-02-24 08:43:52.000</OsReportDateTime>
  <OsReportTimeZone>C</OsReportTimeZone>
  <OsTimeUtc>2012-02-24 14:43:52.000</OsTimeUtc>
  <MilePost>468.8</MilePost>
  <MilePostModifier></MilePostModifier>
  <TypeMove>A</TypeMove>
  <LineSegment>7300</LineSegment>
</OSMessage>
```

Table A.5. HBD Message Schema

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="HBDMessage">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="HBDMessageID" type="xs:positiveInteger"/>
        <xs:element name="AxleSeq" type="xs:positiveInteger"/>
        <xs:element name="HBDSideCode" type="xs:positiveInteger"/>
        <xs:element name="CreateDate" type="xs:dateTime"/>
        <xs:element name="HBDBearingTemp" type="xs:positiveInteger"/>
        <xs:element name="EquipmentInit" type="xs:string"/>
        <xs:element name="EquipmentNumber" type="xs:positiveInteger"/>
        <xs:element name="AxleSide" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Table A.6. Sample HBD Message

```
<HBDMessage>
  <HBDMessageID>12661136</HBDMessageID>
  <AxleSeq>28</AxleSeq>
  <HBDSideCode>1</HBDSideCode>
  <CreateDate>2/24/2012</CreateDate>
  <HBDBearingTemp>29</HBDBearingTemp>
  <EquipmentInit>DTTX</EquipmentInit>
  <EquipmentNumber>751286</EquipmentNumber>
  <AxleSide>L</AxleSide>
</HBDMessage>
<HBDMessage>
  <HBDMessageID>12661136</HBDMessageID>
  <AxleSeq>28</AxleSeq>
  <HBDSideCode>2</HBDSideCode>
  <CreateDate>2/24/2012</CreateDate>
  <HBDBearingTemp>43</HBDBearingTemp>
  <EquipmentInit>DTTX</EquipmentInit>
  <EquipmentNumber>751286</EquipmentNumber>
  <AxleSide>R</AxleSide>
</HBDMessage>
```

Table A.7. Sample HBD Message

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="HBDMessage">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="HBDMessageID" type="xs:positiveInteger"/>
        <xs:element name="AxleSeq" type="xs:positiveInteger"/>
        <xs:element name="HBDSideCode" type="xs:positiveInteger"/>
        <xs:element name="CreateDate" type="xs:dateTime"/>
        <xs:element name="HBDBearingTemp" type="xs:positiveInteger"/>
        <xs:element name="EquipmentInit" type="xs:string"/>
        <xs:element name="EquipmentNumber" type="xs:positiveInteger"/>
        <xs:element name="WheelTrackNumber" type="xs:string"/>
        <!--truck number starting from 'A'
             end of the car - A/B/C/D-->
        <xs:element name="AxleSide" type="xs:string"/>
        <xs:element name="EquipmentAxleNumber" type="xs:positiveInteger"/>
        <!-- sequence # to identify an axle on a
               car/loco based on orientation-->
        <xs:element name="HbdLevelingFactor" type="xs:decimal"/>
        <!--compensates for operating differences in different HBDs-->
        <xs:element name="HbdAxleDistance"  type="xs:decimal"/>
        <!-- distance between consecutive axles
               on a car/locomotive in inches-->
        <xs:element name="HbdTrend"  type="xs:decimal"/>
        <!--  hot box detector trend from previous
                reading to current reading-->
        <xs:element name="KCar" type="xs:decimal"/>
        <!--k value from equipment temperature distribution at current
            HBD, calculated from the quartiles-->
        <xs:element name="ZScore" type="xs:decimal"/>
        <xs:element name="KDifferenceMin" type="xs:decimal"/>
        <!--difference between the first and
              the second largest k value of a car-->
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

REFERENCES

[1] S. Bose, "Query processing on streamed xml data," Ph.D. dissertation, Univ. of Texas at Arlington, Arlington, May 2005.

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

[3] C. C. Aggarwal, *Data Streams: Models and Algorithms (Advances in Database Systems).* Secaucus, NJ: Springer-Verlag New York, Inc., 2006.

[4] N. Polyzotis and M. Garofalakis, "Structure and value synopsis for xml data graphs," in *Proc. VLDB Conference*, 2002.

[5] N. Polyzotis and M. Garofalakis, "Xcluster: Synopses for structured xml content," in *Proc. ICDE Conference*, 2006.

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

[7] L. Fegaras, R. Dash, and Y. Wang, "A fully pipelined xquery processor," in *Proc. XIME-P*, 2006.

[8] R. Dash and L. Fegaras, "Synopsis based load shedding in xml streams," in *Proc. DataX*, 2009.

[9] R. Dash and L. Fegaras, "A load shedding framework for xml stream joins," in *Proc. DEXA*, 2010.

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

[11] R. Dash and L. Fegaras, "Multi-dimensional data stream aggregation with join using stream cubes," in *Proc. EDBT*, submitted for publication.

[12] 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*, 2002, pp. 215–226.

[13] 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*, no. 13(4), pp. 370–383, 2004.

[14] D. Abadi, "Aurora: A new model and architecture for data stream management," *VLDB Journal*, no. 12(2), pp. 120–139, 2003.

[15] S. Zdonik, Stonebraker, M. M., Cherniack, U. Cetintemel, M. Balazinska, and H. Balakrishnan, "The aurora and medusa projects," *IEEE Data Eng. Bull*, no. 26(1), pp. 3–10, 2003.

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

[17] 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*, 2002, pp. 1–16.

[18] (2003) Stream: Stanford stream data management (stream) project. [Online]. Available: http://www-db.stanford.edu/stream

[19] S. Babu and J. Widom, "Continuous queries over data streams," in *Proc. ACM-SIGMOD International Conference on Management of Data*, 2001, pp. 109–120.

[20] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas, "Operator scheduling in data stream systems," pp. 333–353, 2004.

[21] B. Babcock, M. Datar, and R. Motwani, "Load shedding for aggregation queries over data streams," in *Proc. International Conference on Data Engineering*, no. 13(4), 2004, pp. 350–361.

[22] A. Arasu and J. Widom, "Resource sharing in continuous sliding-window aggregates," in *Proc. Thirtieth International Conference on Very Large Data Bases*, 2004, pp. 336–347.

[23] 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*, 2003, pp. 245–256.

[24] S. Babu, K. Munagala, J. Widom, and R. Motwani, "Adaptive caching for continuous queries," in *Proc. 21st International Conference on Data Engineering*, 2005, pp. 118–129.

[25] Q. Jiang, "Data stream management system for mavhomea framework for supporting quality of service requirements in a data stream management system," Ph.D. dissertation, The University of Texas at Arlington, 2005. [Online]. Available: http://itlab.uta.edu/ITLABWEB/Students/sharma/theses/Jia05PHD.pdf

[26] Q. Jiang and S. Chakravarthy, "Data stream management system for mavhome," in *Proc. Annual ACM SIG Symposium On Applied Computing*, 2004, pp. 654–655.

[27] 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.

[28] Q. Jiang and S. Chakravarthy, "A framework for supporting load shedding in data stream management systems,"

UT Arlington," TR CSE-2004-19, 2004. [Online]. Available: http://www.cse.uta.edu/research/publications/Downloads/CSE-2004-19.pdf

[29] B. Kendai and S. Chakravarthy, "Load shedding in mavstream: Analysis, implementation, and evaluation," in *Proc. British National Conference on Databases (BNCOD)*, 2008, pp. 100–112.

[30] E. A. Rundensteiner, L. Ding, Y. Zhu, T. Sutherland, and B. Pielech, *CAPE:A Constraint-Aware Adaptive Stream Processing Engine.* Secaucus, NJ: Springer-Verlag, 2004.

[31] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid, "Scheduling for shared window joins over data streams," in *Proc. VLDB*, Sept. 2003, pp. 297–308.

[32] J. Kang, J. F. Naughton, and S. D. Viglas, "Evaluating window joins over unbounded streams," in *Proc. ICDE*, 2003, pp. 341–352.

[33] L. Ding, N. Mehta, E. A. Rundensteiner, and G. T. Heineman, "Joining punctuated streams," in *Proc. EDBT*, pp. 587–604.

[34] P. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting punctuation semantics in continuous data streams," in *Proc. IEEE Transactions on Knowledge and Data Engineering*, no. 15(3), 2003, pp. 555–568.

[35] T. Sutherland, B. Pielech, and E. A. Rundensteiner, "Adaptive scheduling framework for a continuous query system," Worcester Polytechnic Institute," Technical Report WPI-CS-TR-04-16, 2004.

[36] N. Chaudhry, K. Shaw, and M. Abdelgueifi, *STREAM DATA MANAGEMENT(Advances in Database Systems).* Secaucus, NJ: Springer-Verlag New York, Inc., 2005.

[37] 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.

[38] 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.

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

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

[41] K. Chakrabarti, M. Garofalakis, R. Rastogi, and S. K., "Approximate query processing with wavelets," pp. 199–223, 2001.

[42] N. Alon, P. Gibbons, Y. Matias, and M. Szegedy, "Tracking joins and self joins in limited storage." in *Proc. ACM PODS Conference*, 1999.

[43] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi, "Processing complex aggregate queries over data streams," in *Proc. SIGMOD*, 2002.

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

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

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

[47] G. Manku, S. Rajagopalan, and B. Lindsay, "Random sampling for space efficient computation of order statistics in large datasets," in *Proc. ACM SIGMOD*, 1999, pp. 100–112.

[48] G. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proc. VLDB Conference*, 2002.

[49] C. Aggarwal, J. Han, and P. Wang, J.and Yu, "A framework for clustering evolving data streams," in *Proc. VLDB Conference*, 2003.

[50] 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.

[51] P. Gibbons and Y. Mattias, "New sampling-based summary statistics for improving approximate query answers," in *Proc. ACM SIGMOD Conference*, 1998.

[52] B. Babcock, M. Datar, and M. R, "Sampling from a moving window over streaming data," in *Proc. ACM SIAM Symposium on Discrete Algorithms*, 2002.

[53] C. Aggarwal, "On biased reservoir sampling in the presence of stream evolution," in *Proc. VLDB*, 2006.

[54] G. Cormode, M. Garofalakis, and D. Sacharidis, "Fast approximate wavelet tracking on streams," in *Proc. EDBT Conference*, 2006.

[55] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss, "Surfing wavelets on streams: One pass summaries for approximate aggregate queries," in *Proc. VLDB Conference*, 2001.

[56] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss, "One-pass wavelet decompositions of data streams," in *IEEE TKDE*, no. 15(3).

[57] S. Guha, N. Koudas, and K. Shim, "Data-streams and histograms," in *Proc. ACM STOC Conference*, 2001.

[58] C. C. Aggarwal and P. S. Yu, *A SURVEY OF SYNOPSIS CONSTRUCTION IN DATA STREAMS.* Springer-Verlag New York, Inc., 2006.

[59] B. Ludascher, P. Mukhopadhyay, , and Y. Papakonstantinou, "A transducer-based xml query processor," in *Proc. VLDB*, 2002, pp. 227–238.

[60] A. Gupta and D. Suciu, "Stream processing of xpath queries with predicates," in *Proc. ACM SIGMOD*, 2003, pp. 419–430.

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

[62] 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*, no. 3(1), 2010, pp. 1267–1278.

[63] A. Das, J. Gehrke, and M. Riedwald, "Approximate join processing over data streams," in *Proc. ACM SIGMOD*, 2003, pp. 40–51.

[64] U. Srivastava and J. Widom, "Memory-limited execution of windowed stream joins," in *Proc. VLDB*, 2004.

[65] 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.

[66] 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.

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

[68] M. Hong, A. Demers, J. Gehrke, C. Koch, M. Riedewald, and W. White, "Massively multi-query join processing," in *Proc. SIGMOD*, Beijing, China, 2007, pp. 11–14.

[69] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava, "Multiple aggregations over data streams," *Proc. 2005 ACM SIGMOD international conference on Management of data*, pp. 373–378, June 14-16 2005.

[70] N. Tatbul and S. Zdonik, "Window-aware load shedding for aggregation queries over data streams," in *Proc. 2006 International Conference on Very Large Databases*, 2006, pp. 799–810.

[71] S. Chaudhuri and K. Shim, "Including group-by in query optimization," in *Proc. 20th International Conference on Very Large Databases*. Morgan Kaufmann, 1994, pp. 354–366.

[72] 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*, 2006, pp. 182–192.

[73] T. M. Tran and B. S. Lee, "Transformation of continuous aggregation join queries over data streams," in *Proc. SSTD*, 2007, pp. 330–347.

[74] W. P. Yan and P. Larson, "Performing group-by before join," in *Proc. International Conference Data Engineering, IEEE Computer Society*, 1994, pp. 89–100.

[75] W. P. Yan and P. Larson, "Eager aggregation and lazy aggregation," in *Proc. 21st International Conference on Very Large Databases*. Morgan Kaufmann, pp. 345–357.

[76] V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Implementing data cubes efficiently," in *Proc. 1996 ACM SIGMOD international conference on Management of data*, Montreal, Quebec, Canada, pp. 205–216.

[77] J. Han, J. Pei, G. Dong, and K. Wang, "Efficient computation of iceberg cubes with complex measures," in *Proc. 2001 ACM SIGMOD international conference on Management of data*, May, pp. 1–12.

[78] Y. Zhao, P. M. Deshpande, and J. F. Naughton, "An array-based algorithm for simultaneous multidimensional aggregates," *Proc. 1997 ACM SIGMOD international conference on Management of data*, pp. 159–170, May 1997.

[79] A. Cuzzocrea, "Cubing algorithms for xml data," in *Proc. 2009 20th International Workshop on Database and Expert Systems Application*, 2009, pp. 407–411.

[80] 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," no. 18(2), pp. 173–197, 2005.

[81] Y. Cai, D. Clutterx, G. Papex, J. Han, and M. Welgex, "Maids: Mining alarming incidents from data streams," in *Proc. ACM SIGMOD*, 2004, pp. 919–920.

[82] Y. Chen, G. Dong, J. Han, J. Pei, B. Wah, and J. Wang, "Olaping stream data: Is it feasible," in *Proc. Workshop on Research Issues in Data Mining and Knowledge Discovery*, 2002, pp. 53–58.

[83] W. S. Yang and W. S. Lee, "On-line evaluation of a data cube over a data stream," in *Proc. 8th conference on Applied computer science*, 2008, pp. 373–378.

[84] A. Cuzzocrea, "Cams: Olaping multidimensional data streams efficiently," in *Proc. DaWaK*, 2009, pp. 48–62.

[85] S. Berchtold, C. Bohm, and H. P. Kriegel, "The pyramid-technique: Towards breaking the curse of dimensionality," in *Proc. ACM SIGMOD*, 1998, pp. 142–153.

[86] G. Liang, L. Runheng, J. Yan, and J. Xin, "Compressed streamcube: Implementation of compressed data cube in dsms," in *Proc. International Conference on Computing, Control and Industrial Engineering*, June 14-16 2010, pp. 345–349.

[87] 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*, San Diego, California, United States, August 1999, pp. 223–232.

[88] D. Megginson. (2002) Simple api for xml. [Online]. Available: http://www.saxproject.org/.

[89] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Simeon, "Statix: Making xml count," in *Proc. ACM SIGMOD*, 2002.

[90] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton, "Estimating the selectivity of xml path expressions for internet scale applications," in *Proc. VLDB*, 2001.

[91] N. Polyzotis, M. Garofalakis, and Y. Ioannidis, "Approximate xml query answers," in *Proc. ACM SIGMOD Conference*, 2004.

[92] W. Wang, H. Jiang, H. Lu, and J. X. Yu, "Containment join size estimation: Models and methods," in *ACM SIGMOD*, 2003.

[93] L. Lim, M. Wang, S. Padmanabhan, J. Vitter, and R. Parr, "Xpathlearner: An on-line self-tuning markov histogram for xml path selectivity estimation," in *Proceedings of the VLDB*, 2002.

[94] N. Polyzotis, M. Garofalakis, and Y. Ioannidis, "Selectivity estimation for xml twigs," in *Proc. ICDE*, 2004.

[95] Y. Wu, J. M. Patel, and H. Jagadish, "Estimating answer sizes for xml queries," in *Proc. EDBT*, 2002.

[96] 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.

[97] M. Ley, "Dblp xml records." [Online]. Available: http://www.informatik.uni-trier.de/ ley/db/

[98] B. Chandramouli and J. Yang, "End-to-end support for joins in large-scale publish/subscribe systems," in *Proceedings of the VLDB*, August 2008.

[99] A. Ayad, J. Naughton, S. Wright, and U. Srivastava, "Approximating streaming window joins under cpu limitations," in *Proceedings of the ICDE*, 2006.

[100] G. Cormode, V. Shkapenyuk, D. Srivastava, and B. Xu, "Forward decay: A practical time decay model for streaming systems," in *ICDE*, pp. 138–149.

[101] "Stylus studio - xml editor, xml data integration, xml tools, web services and xquery." [Online]. Available: http://www.stylusstudio.com/

[102] 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.

[103] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, , and R. Busse, "Xmark: A benchmark for xml data management," *Proc. VLDB*, pp. 974–985, 2002.

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

[105] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. Ullman, "Computing iceberg queries efficiently," in *Proc. VLDB Conference*, 1998, pp. 299–310.

[106] D. Yang, A. Shastri, E. A. Rundeinsteiner, and M. O. Ward, "An optimal strategy for monitoring top-k queries in streaming windows," in *Proc. EDBT*, 2011.

[107] G. Das, D. Gunopulos, N. Koudas, and N. Sarkas, "Adhoc top-k query answering for data streams," *Proc. VLDB*, 2007.

[108] H. Lam and T. Calders, "Mining top-k frequent items in a data stream with flexible sliding windows," *Proc. KDD*, July 2010.

[109] A.Metwally, D. Agrawal, and A. E. Abbadi, "Efficient computation of frequent and top-k elements in data streams," *Proc. ICDT*, June 2005.

[110] L. Wang, Y. K. Lee, and K. Ryu, "Supporting top-k aggregate queries over unequal synopsis on internet traffic streams," *Proc. APWeb 2008*, pp. 590–600, 2008.

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

[112] 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*, pp. 173–178, 2003.

[113] I. Ilyas, G. Beskales, and M. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM Computing Surveys*, vol. 40, no. 4, p. 425425, 2008.

[114] L. Golab, D. DeHaan, A. Lopez-Ortiz, and E. Demaine, "Finding frequent items in sliding windows with multinomially-distributed item frequencies," *16th International Conference on Scientific and Statistical Database Management (SSDBM 2004)*, p. 425425, 2004.

[115] S. Paparizos, S. Al-Khalifa, H. V. Jagadish, L. Lakshmanan, A. Nierman, D. Srivastava, and Y. Wu, "Grouping in xml," *EDBT Workshops*, pp. 128–147, 2002.

[116] W. He, L. Fegaras, and D. Levine, "Indexing and searching xml documents based on content and structure synopses," *Proc. BNCOD*, pp. 58–69, 2007.

[117] P. Tucker, K. T. V. Papadimos, and D. Maier, "Nexmark – a benchmark for queries over data streams," 2002. [Online]. Available: http://www.cse.ogi.edu/dot/niagara/NEXMark

[118] F. Yu and W. Shan, "Compressed data cube for approximate olap query processing," *J. Computer Sc. Tech*, vol. 17(5), pp. 625–635, 2002.

BIOGRAPHICAL STATEMENT

Ranjan Dash was born in Cuttack, India. He received his B.S. and M.S. degrees in Civil Engineering from NIT (Rourkela), India. He received his M.S. and Ph.D. degrees in Computer Engineering from The University of Texas at Arlington in 2001 and 2012 respectively.