2011-12-18

Continuously Providing Approximate Results under Limited Resources: Load Shedding and Spilling in XML Streams

Mingzhu Wei
Worcester Polytechnic Institute

Continuously Providing Approximate Results under Limited Resources: Load Shedding and Spilling in XML Streams

by

Mingzhu Wei

A Dissertation
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy
in
Computer Science
by

Nov. 15, 2011

APPROVED:

Abstract

Because of the high volume and unpredictable arrival rates, stream processing systems may not always be able to keep up with the input data streams, resulting in buffer overflow and uncontrolled loss of data. To continuously supply online results, two alternate solutions to tackle this problem of unpredictable failures of such overloaded systems can be identified. One technique, called load shedding, drops some fractions of data from the input stream to reduce the memory and CPU requirements of the workload. However, dropping some portions of the input data means that the accuracy of the output is reduced since some data is lost. To produce eventually complete results, the second technique, called data spilling, pushes some fractions of data to persistent storage temporarily when the processing speed cannot keep up with the arrival rate. The processing of the disk resident data is then postponed until a later time when system resources become available. This dissertation explores these load reduction technologies in the context of XML stream systems.

Load shedding in the specific context of XML streams poses several unique opportunities and challenges. Since XML data is hierarchical, subelements, extracted from different positions of the XML tree structure, may vary in their importance. Further, dropping different subelements may vary in their savings of storage and computation. Hence, unlike prior work in the literature that drops data completely or not at all, in this dissertation we introduce the notion of structure-oriented load shedding, meaning selectively some XML subelements are shed from the possibly complex XML objects in the XML stream. First we develop a preference model that enables users to specify the relative importance of preserving different subelements within the XML result structure. This transforms shedding into the problem of rewriting the user query into shed queries that return approximate answers with their utility as measured by the user preference model. Our optimizer finds the appropriate shed queries to maximize the output utility driven by our structure-based preference model under the limitation of available computation resources. The experimental results demonstrate that our proposed XML-specific shedding solution consistently achieves higher utility results compared to the existing relational shedding techniques.

Second, we introduces structure-based spilling, a spilling technique customized for XML streams by considering the spilling of partial substructures of possibly complex XML elements. Several new challenges caused by structure-based spilling are addressed. When a path is spilled, multiple other paths may be affected. We categorize varying types of spilling side effects on the query caused by spilling. How to execute the reduced query to produce the correct runtime output is also studied. Three optimization strategies are developed to select the reduced query that maximizes the output quality. We also examine the clean-up stage to guarantee that an entire result set is eventually generated by producing supplementary results to complement the partial results output earlier. The experimental study demonstrates that our proposed solutions consistently achieve higher quality results compared to the state-of-the-art techniques.

Third, we design an integrated framework that combines both shedding and spilling policies into one comprehensive methodology. Decisions on the choice of whether to shed or spill data may be affected by the application needs and data arrival patterns. For some input data, it may be worth to flush it to disk if a delayed output of its result will be important, while other data would best directly dropped from the system given that a delayed delivery of these results would no longer be meaningful to the application. Therefore we need sophisticated technologies capable of deploying both shedding and spilling techniques within one integrated strategy with the ability to deliver the most appropriate decision customers need for each specific circumstance. We propose a novel flexible framework for structure-based shed and spill approaches, applicable in any XML stream system. We propose a solution space that represents all the shed and spill candidates. An age-based quality model is proposed for evaluating the output quality for different reduced query and supplementary query pairs. We also propose a family of four optimization strategies, OptF, OptSmart, HiX and Fex. OptF and OptSmart are both guaranteed to identify an optimal solution of reduced and supplementary query pair, with OptSmart exhibiting significantly less overhead than OptF. HiX and Fex use heuristic-based approaches that are much more efficient than OptF and OptSmart.

Acknowledgements

I am extremely grateful to many people for all the guidance and help I have received throughout the period of my Ph.D. studies.

My deepest gratitude is to Prof. Elke A. Rundensteiner. I am grateful to her for giving constant encouragement and support throughout my graduate studies. I appreciate all her contributions of time and ideas to make my Ph.D. experience stimulating. The enthusiasm she has for her research was contagious and motivational for me. I am also thankful for the excellent example she has provided as a successful woman professor. I would like to express my gratitude to my co-advisor Prof. Murali Mani. I thank Murali for his insightful comments and support on every stage of my research. Murali has spent enormous amount of time discussing with me and giving constructive feedback on my papers.

My thanks go to the members of my Ph.D. committee, Prof. Robert Lindeman and Dr. Jerome Simeon, who provided valuable feedback and suggestions to my comprehensive examination, my dissertation proposal and dissertation. Their impact helped to improve the presentation and contents of this dissertation.

I would like to express my gratitude to my mentor Ismail Ari during my internship at HP labs. I feel truly lucky to have him as my mentor. I thank him for his understanding, advice and support during my internship.

I also would like to thank the current and the former members of the Database Systems Research Group (DSRG) at Worcester Polytechnic Institute with whom I have interacted during the course of my graduate studies. Particularly, I would like to acknowledge Luping Ding, Yali Zhu, Song Wang, Hong Su, Mo Liu, Di Wang, Ming Li, Ling Wang, Venkatesh Raghavan, Bin Liu, Di Yang, Karen Works, Abhishek Mukerji (and many others) for the many valuable discussions that helped me understand my research area better.

I thank the wonderful professors in the Computer Science department for both their serious lectures and casual chats. I thank the system support staff in our department for providing a well-maintained computing environment and utilities for my research needs.

I would like to thank my husband Zaixian Xie for his love, support and the never-ending encouragement. My parents receive my deepest gratitude and love for their dedication and the many years of support during my studies. I would like to express my gratitude to my parents-in-law for their selfless help on taking care of my son when he was very young. The final but not the least thank you goes to my son, Hansen. His sweet smile brings me a lot of joy during tough times in the Ph.D. pursuit.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 General Concepts of XML Stream Processing

Recent years have witnessed a rapidly increasing attention on streaming database systems [1, 2, 6, 9, 17, 34, 50, 64] because of the development of web and network techniques. Different from traditional database systems with statically stored data and one-time queries, in a streaming database, data arrives on-the-fly. User queries are generally long-running or continuous, and the results of the queries are also in the format of output streams. This type of query is generally referred to as a continuous query.

Continuous queries significantly differ from traditional static queries in following aspects.

1. Data availability. For traditional relational queries, the data is known *a priori* and is persistently stored on disk. However, the stream data arrives at the system via some network link in a never ending stream. For instance, monitoring applications process data streams from sensor networks to monitor storehouse temperature or road traffic. In network analysis applications, streams of network packets are sent to the system to detect intrusions. In these scenarios, system has no data stored

before new data arrives.

2. Result generation. Generation of query results for static queries is driven by a pull-based execution strategy. However, when stream data arrives on-the-fly, the query processing will be driven by the data and will thus produce results in a push-based fashion.

Due to the proliferation of XML data in web services, there is also a surge in XML stream applications [15, 16, 22, 29, 32, 33, 44, 51, 52, 55]. For instance, a message broker routes the XML messages to interested parties [29]. In addition, message brokers can also perform message restructuring or backups. For example, in an online order handling system [16], suppliers can register their available products with the broker. The broker will then match each incoming purchase order with the subscription and forward it to the corresponding suppliers, possibly in a restructured format at the request of the suppliers. Other typical applications include XML packet routing [55], selective dissemination of information such as personalized newspaper delivery [4], and XML monitoring systems [51] for online auctions.

In XML streams, it is possible that an XML tuple (the basic unit to generate output result) is split into many small pieces. Thus the incoming data is entering the system at the granularity of a continuous stream of tokens [22, 44] or fragments [25], instead of complete tree structured XML element nodes. Different from relational stream systems, XML stream processing experiences new challenges.

1. Stream data arrives at the granularity of tokens or fragments. Since a single sequential scan of input data is only allowed, the engine has to either extract relevant tokens to form XML elements or to compose XML fragments to complete XML structures.

2. We need to conduct dissecting, restructuring, and assembly of complex nested XML elements specified by query expressions, such as XQuery [65].

1.2 Motivation for Structure-based Shedding and Spilling

After giving a brief introduction about XML stream processing, now we motivate structure-based shedding and spilling for XML streams. For most stream applications, immediate online results are required. However, stream applications are often characterized by push-based data sources in which the arrival rates can be high and unpredictable. When the arrival rate is very high, stream processing systems may not always be able to keep up with the input data streams–resulting in buffer overflow and uncontrolled loss of data. Since such overload situations are usually unforeseen and immediate attention is vital, adapting the system capacity to the increased load by adding more resources or distributing computation to multiple nodes may not be feasible or economically meaningful. In this case, the only immediate solution is to reduce some of the load. Load shedding and load spilling are two load reduction techniques proposed to solve the issue of insufficient system resources to keep up with the processing of the data stream. Load shedding is a strategy for solving this overflow problem by discarding a subset of the input data (tuples) without processing–whenever the rate of processing data is not able to keep pace with the input rate [10, 21, 28, 59]. Load spilling flushes some subset of the input steam to disks temporarily. The processing of the disk resident data is postponed until a later time, for instance, when there is a lull in the input stream.

We note that shedding applied to complex data types, such as XML streams, brings new opportunities and challenges due to the complex nested nature of the XML element structures. To generate as many output results as we can, we now instead propose to throw away some sub patterns from an XML query result tree–which the initial query specification was supposed to extract. This may result in savings in the processing for each output result, however this is at the cost of reducing the accuracy of the output structure itself. How to assure a certain accuracy while still returning as much output as possible is a challenging open issue.

In some applications, shedding may not be applicable since complete results may still be required to be generated or at least retrievable at some later time. For instance, in network intrusion detection systems, we need to analyze the packet information to detect potential attacks. If some packets are dropped, the thrown packets may contain the information related to the attack. In this case, throwing packets directly may lead to a later failure to detect some attacks–possibly in a post-analysis process. Thus load shedding techniques may not be suitable for such applications.

For the applications that require complete results, we would instead deploy a structure-based spill technique, namely, to flush some sub patterns from an XML query result to disks temporarily. Later when system resources become available again, we can continue to finish the processing of the remaining disk resident data to produce the supplementary output. Here we propose the notion of structure-based spilling in XML streams. We aim to provide solutions for structure-based spilling that produce partial results, supplemented later by refreshed delta result structures as to maximize the output data utility.

Last but not least, we develop an integrated load reduction framework that combines both structure-based shedding and spilling policies within one uniform manner. The intuition is that some input data may be worthwhile waiting for, as even a delayed output of a result will be important so we temporarily spill data, that can be salvaged by a later un-spill). Otherwise we may as well directly shed the input data from the system–given that a delayed delivery of any result produced based on this input data at a later time would no longer be of relevance to the application. Our goal is to design a carefully calibrated multi-method framework that successfully applies both technologies to achieve maximal effectiveness in processing input streams while serving the needs of the applications best.

1.3 State-of-the-Art Load Shedding and Spilling Techniques

1.3.1 Load Shedding Techniques

In streaming systems, load shedding has been considered an effective method for trading off performance with accuracy [21, 49, 56, 59, 61]. Currently most load shedding techniques have been developed for relational streams. Load shedding on streaming data was first proposed in the Aurora system [59]. This work introduces two types of load shedding: random and semantic load shedding. Based on the analysis of the loss/gain rate, the random load shedding strategy will determine the amount of tuples to shed to guarantee that the remainder of the input can indeed be handled. For semantic drop, they assume that different tuple values may vary in terms of their utility to the application. A frequency-based stream model [21] is proposed for sliding window joins. In this model, each join value has a fixed frequency of the data streams and hence drops tuples based on their popularity. An age-based stream model is proposed in [56]. In this age-based model, every tuple in the data stream is confined to follow an aging process such that the expected join multiplicity of a tuple is dependent on its arrival time. A load shedding approach for join processing is proposed based on this age-based stream model in [56] . An adaptive CPU load shedding approach [28] is provided for window stream joins that follows the selective processing tuple methodology in windows. However, for XML streams, we must consider the complexity as well as importance of XML result structures in order to make reduced query decisions.

1.3.2 Data Spilling Techniques

In many cases, long running queries may need to produce complete result sets, even though the query system may not have sufficient resources for the query workload at a particular time. As an example, decision support applications rely on complete results to eventually apply complex and long-ranging historic data analysis, i.e., quantitative analysis. One viable solution to address the problem of run-time main memory shortage while satisfying the needs of complete query results is to push memory resident states temporarily into disks when memory overflow occurs. Such solutions have been discussed in XJoin [63], Hash-Merge Join [48] and MJoin [19]. These solutions aim to ensure a high runtime output rate as well as the completeness of query results for a query that contains a single operator. The processing of the disk resident states, referred to as state cleanup, is delayed until a later time when more resources become available. The spilling solutions for query plan with multiple query operators are proposed in [43] where data spilling from one operator can affect other operators in the same pipeline. We could directly apply the above techniques from the literature to coarse-grained spilling in XML, namely, to spilling complete topmost elements to disk; however, such coarse-grained spilling misses the XML-specific opportunities for spilling. In this dissertation, we instead focus on the fine-grained XML-specific structural spilling approach.

Generating partial XQuery results is discussed in [54] when the output is requested in Internet applications. However, they only address how to produce partial results when only partial data is available. They did not consider the problem of resource management under limited resources scenario in general, nor the specifics of producing partial results in the XML stream context when the output from one operator is missing due to spilling some patterns.

1.4 Research Focus

In this dissertation, we explore structure-based shedding and spilling for XML streams. The overall goal of this dissertation is to develop load reduction techniques including structural shedding and spilling to optimize the production of output results for XML streams. The dissertation is focused on the following three topics: 1) Structure-based shedding for XML streams, 2) Structure-based spilling for XML streams, and 3) An integrated framework with a hybrid structure-based drop and flush approach for XML streams.

1.4.1 Structure-based Shedding for XML Streams

The first dissertation goal is structural shedding for XML streams which selectively drops XML subelements to achieve a high processing speed.

Now let us look at a concrete example. Consider an online store, customers may have periods of heavy usage during some promotions or near holidays. The online store would receive huge numbers of order from customers during these times. When the processing capacity is not sufficient to keep up with the data arrival rate, the data will accumulate in the buffer resulting in an overflow. In this case, we have to either drop some data or improve the processing speed. We consider the topmost “transaction” element in the schema a basic unit based on which we can generate results. However, dropping complete “transaction” elements means that we may lose important information. In this scenario, dropping unimportant but resource-intensive subelements may be more meaningful to output applications compared to the complete-tuple-granularity shedding. We call this type of “element” granularity drop *structural shedding* since it changes the structure of query results.

Let us consider the online store query in Figure 2.1. This query returns the item list and contact information including telephone, email and address when customers spend more than 100 dollars. To process as many transaction elements as possible, consumers of the query result may prefer to selectively obtain partial yet important content as result while dropping less important subelements in each transaction tuple. In this case we may choose to drop "addr" information for two reasons.

1. "addr" element is much more complex than "email," as shown in the schema shown in Figure 1.2. This means we process more tokens for each single "addr" element
2. "addr" element may be "optional" to output consumers because "email" may be the more likely means of contacting customers

By dropping the "addr" element, several savings arise. First, we do not need to process "addr" element from the input tokens. In this case, we bypass the processing of tokens from "<addr>" to "</addr>." Second, we no longer need to buffer "addr" element during processing. Thus the buffering costs for "addr" element are saved. Note here this shedding can be achieved by removing the "addr" element from the initial query. We call the new reduced query *shed query*.

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

Figure 1.1: Query Q1

There are many options to drop subelements from a given query. However, different shed queries vary in their importance and their processing costs. Hence the correct choice of appropriate shed queries raises many challenges. First, what model do we employ

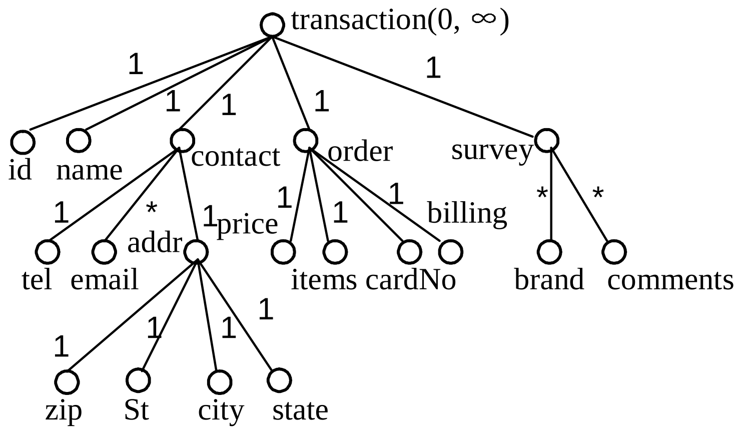


Figure 1.2: The Schema Definition for Q1

to specify the importance of each subelement? Second, after generating different shed queries, how can we estimate the cost of these shed queries at runtime? Third, which of the potential shed queries should be chosen to obtain maximum output utility? Our solution tackles these challenges using a three-pronged strategy. One, we propose a preference model for XQuery to enable output consumers to specify the relative *utility* (preference) of preserving different sub-patterns in the query. Two, we develop a cost model to estimate the processing cost for the candidate shed queries. Three, we transform the shed query decision problem into an optimization problem, and propose two solutions. The main goal of our shedding technique is to maximize output utility given the stream input rate and limited computational resources.

Contributions. This part of the dissertation work contributes to research in load shedding in XML streams in the following ways:

1. First, a structure-based preference model is proposed that uniquely exploits the relative importance of different sub-patterns in XML query results.

2. Second, we formulate the shedding problem as an optimization problem to find

the appropriate shed queries to maximize the output utility based on our structure-based preference model and the estimated cost derived from our cost model for XML streams.

3. Third, to solve the shedding problem, we develop two algorithms, OptShed and FastShed. OptShed guarantees to find an optimal solution however at the cost of an exponential complexity. FastShed achieves a close-to-optimal result in a wide range of cases with much smaller search costs than OptShed.

4. Fourth, we propose a simple yet elegant in-automaton shedding mechanism by suspending the appropriate states in the automaton-based execution engine for XML streams, in order to drop data early (and efficiently).

5. Finally, we provide a thorough experimental evaluation that demonstrates that our approach maximizes the utility while keeping CPU costs under the available system capacity.

1.4.2 Structure-based Spilling for XML Streams

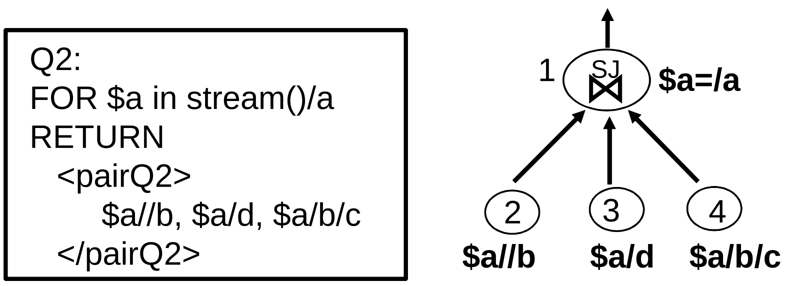
The second dissertation goal is to explore structural spilling in XML streams. We aim to provide solutions for structure-based spilling that produce partial results, supplemented later by refreshed delta result structures so to maximize the output data utility. To the best of our knowledge, there is no prior work on exploring structure-based spilling. We now describe the practicability of structure-based spilling via concrete application scenarios below.

Example 1. In online auction environments, sellers may continuously start new auctions. When customers search for “SLR cameras,” all matching cameras and their product information should be returned. Some key portions of the results, such as price and customer ratings, will be displayed first, which aid customers in making decisions. Many consumers tend to use a two-stage process to reach their decisions [31] instead of inspecting complete product information immediately. Consumers typically identify a subset of the most promising alternatives based on the displayed results. Other product attributes, such as sizes, and features, are often evaluated later after consumers have identified their favorite subsets. When system resources are limited, the query engine may spill unimportant attributes to disk while producing partial results containing key information such as price and customer ratings.

Example 2. In network intrusion detection systems, XML streaming data may come from different nodes of the wide-area network. We need to analyze the incoming packet information to detect potential attacks. If some packets are dropped, the discarded packets may contain the information related to the attack. In this case, dropping packets directly may lead to a later failure to detect and understand the ins and outs of attacks. Instead, pushing unimportant fractions of data to disks temporarily when system resources are limited can avoid such problem.

Example 3. FaceBook users may edit their personal profiles and send messages to their friends at any time. Status updates, composed of possibly nested structures including updates from friends, recent posts on the wall and news from the subscribed group, are generated continuously. However, different users might be interested in specific primary updates. For instance, a college student wants to make new friends. He wants to be notified when his friends add new friends. A girl who likes seeing pictures of her friends hopes to get notified as soon as her friends update their albums. When the system resources are limited, it may be favorable to delay the output of unimportant updates and instead only report "favorite updates" to the end users.

Let us look at a structural spilling example. Query Q2 and its plan are shown in Figure 1.3. Query Q2 returns three path expressions, $\$a//b$, $\$a/d$ and $\$a/b/c$. The plan conducts structural joins on the binding variable $\$a$ and these three path expressions. In



```
Q2:
FOR $a in stream()/a
RETURN
  <pairQ2>
     $a//b, $a/d, $a/b/c
  </pairQ2>
```

(a) **Query Q2** (b) **Query Plan**

Figure 1.3: Query Q2 and Its Plan

this work, we assume any path and any number of paths in the query can be spilled to disk when the system cannot keep up with the arrival rate. Assume the path $/a//b$ is chosen to be spilled, i.e., all b elements on path $/a//b$ are flushed to disk [1]. Note that data corresponding to paths 2 and 4 in the plan is actually affected (as side effect) by such spilling. For each output tuple (e.g., <pairQ2> in Q2), partial result structures are produced since both b and c elements are missing. In this case, several savings arise. First, since complete b elements are pushed to disk from the token stream, we do not need to bother to extract "c" elements from the input at this time. In other words, we bypass the processing of tokens from "<c>" to "</c>." Second, we no longer need to conduct structural joins between $\$a$ and $\$a//b$ nor between $\$a$ and $\$a/b/c$. Henceforth, we refer to the user query after spilling has been applied as *reduced query* and the early output produced by it as *reduced output*.

Such structural-based spilling brings new challenges that do not exist in relational streams. There are many options to spill paths from a given query. Different reduced queries may vary in their processing costs and output quality. Hence the correct choice of appropriate reduced query raises many issues: 1) which additional paths in the query are affected by spilling a particular path; 2) how to estimate the cost of alternative reduced

[1]Terms spill and flush are synonymous and refer to the process of pushing data to disk. We use spill and flush interchangeably in this dissertation.

queries as well as the partial result quality; and 3) which potential reduced query should be chosen to obtain maximum output quality. We tackle these challenges using a three-pronged strategy. One, we examine how to execute reduced queries given varying spilling effects on the query. Two, we provide metrics for measuring the quality and cost of the alternative reduced queries. Three, we transform the reduced query selection problem into an optimization problem, namely, the design of the reduced query that maximizes output quality. Our goal is to generate as many high-quality results as possible given limited resources.

In addition, to eventually produce entire yet duplicate-free result set, we need to generate supplementary results correctly at a later time when the system has sufficient computing resources. For this, we design an output model to match supplementary “delta” structures with partial result structures produced earlier. To generate supplementary results, we determine what extra data to flush to disk to guarantee that the entire result set can be produced.

Contributions. This part of the dissertation work contributes to research in load spilling in XML streams in the following ways:

1. A general framework to address structure-based spilling which can be applied in any XML stream system is proposed.

2. The structure-based spilling problem is formulated into an optimization problem, namely, to find the reduced query that maximizes the output quality based on our structure-based quality and cost model for XML streams.

3. The spilling effect on different paths in the query for a particular spilled path is examined. How to execute the reduced query to produce the correct runtime output is studied.

4. A family of three optimization strategies, OptR, OptPrune and ToX, is proposed to

maximize the output quality for structural spilling. Both OptR and OptPrune are guaranteed to identify an optimal reduced query, with OptPrune exhibiting significantly less overhead than OptR. Using a heuristic-based approach, ToX is much more efficient than OptR and OptPrune.

5. A complementary output model is proposed, that enables us to match supplementary "delta" result structures with partial output produced earlier.

6. The experimental results demonstrate that our optimization strategies consistently achieve higher quality results compared to the state-of-the-art techniques.

1.4.3 An Integrated Framework For Structure-based Shedding and Spilling

When the arrival rates are high and unpredictable, load shedding and spilling are two load reduction techniques proposed to solve the issue of insufficient system resources to keep up with the processing of the stream. However, the state-of-the-art literature has so far overlooked that critical disadvantages exist for both the shed as well as the spill techniques. On the one hand, shedding data means that partial output is lost forever. In addition, dropped data may lead to blocked output, especially when there is a lull in the input. On the other hand, spilling makes the strong assumption that system resources will be ample to process all disk-resident data sooner or later. However, this ignores the fact that in some situations, e.g., network monitoring applications, the data arrival rate of traffic data may remain extremely high for extended periods of time. Huge volumes of data may end up being collected and pushed to disk for archival, wasting CPU resources on the archival and data preparation process. Worst yet, the spilled data may become obsolete before there ever is any opportunity to bring it back into main memory to take advantage of it. This wastes precious resources at a time when instead we should be devoting all resources to pushing out the most critical results in time. Therefore, in some circumstances, neither a strict shed nor a strict spill strategy will be satisfactory, especially in scenarios when the latency of output affects the output quality. Some input data may be worthwhile waiting for, as even a delayed output of a result will be important, thus warranting a temporary spill, that can be salvaged with a later unspilling. While other data would best directly be shed from the system given that a delayed delivering of results would no longer be of relevance to the application resulting in an unnecessary wastage of processing resources. In short, *there is an urgent need for a technology at the middle ground capable of deploying both shedding and spilling techniques within one integrated strategy with the ability to deliver the most appropriate decision customers need for each specific circumstance*.

Motivating Application Scenarios. We now describe the importance and relevance of such an integrated strategy via concrete application scenarios.

In online auction environments where sellers continuously start new auctions, fraud detection is critically important. Fraudulent sellers may use unapproved payment services, such as an unapproved escrow service. For instance, after we detect that a seller uses an escrow service other than the approved www.escrow.com, we should report the seller as fraudster in the output. A fraud detection query and its plan are shown in Figure 1.4 and Figure 1.5 respectively. This query returns three path expressions, $\$a/seller/ID$, $\$/a/bidder/tel$ and $\$a/bidder/price$. The plan conducts structural joins on the binding variable $\$a$ and these three path expressions. Let us assume any query path can be shed or spilled to disk. Assume we can shed or spill one or more query paths. When the system is overloaded, we can choose one or more paths from three query paths $\$a/seller/ID$, $\$/a/bidder/tel$ and $\$a/bidder/price$ to shed or spill. Which paths among them are chosen to be shed permanently versus being spilled to achieve highest output quality is extremely critical in achieving user satisfaction.

```
Fraud detection query:
    FOR $a in stream()/list/auction
    WHERE ($a/seller/payment [contains(., "escrow service")])
        and ($a/seller/payment[ not(contains(., "Escrow.com")])
    RETURN <pairQ1>
                    $a/seller/ID, $a/bidder/tel, $a/bidder/price
                </pairQ1>
```

Figure 1.4: Fraud Detection Query

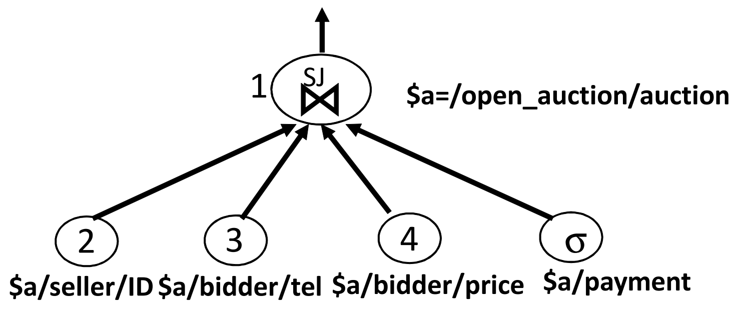


Figure 1.5: Plan for Fraud Detection Query

The naive approach would be to apply the existing algorithms for optimizing either shedding or spilling decisions separately. The shed optimizer would pick the substructures to shed to achieve the highest output quality. For instance, for the fraud detection query, shedding $\{\$a/bidder/price, \$a/bidder/tel\}$ is optimal since the quality for the partial output is the highest. While saying the reduced query can be processed with the given system resources, a spill optimizer on the other hand may choose the reduced query spilling $\{\$a/\ seller/ID$,$\$a/bidder/tel\}$ as the optimal spill candidate. Clearly spilling comes with higher processing costs compared to shedding the same substructure because spilling data to disk comes with the additional overhead of having to execute the disk spill. Therefore this naive approach would ultimately picks the optimal shed solution from the shed optimizer.

Assuming we indeed had the optimal pure shed and pure spill solutions, then another possible solution maybe instead choose some substructures to shed from the optimal shed solution and other substructures to spill from the optimal spill solution. For example, in Q1, we may pick path $\$a/bidder/price$ from the shed solution and the path $\$a/bidder/tel$ from the spill solution. Let us call this composed solution $\{\$a/bidder/price^D$, $\$a/bidder/tel^P\}$ a *fusion candidate* since such a candidate may be a mixture of shed and spill decisions. Here we use a superscript to indicate the action designated for each substructure. D indicates shed and P indicates spill. However, we don't know whether this fusion candidate is the best or even a good solution for a given arrival pattern and available resources in our environment. For this, we would need to compare this particular fusion candidate against other candidates. Instead of conducting such an ad-hoc approach, we clearly need a methodological approach towards tackling this fusion candidate design and fusion candidate selection problem efficiently yet correctly.

Such fine-grained fusion candidates raise many technical challenges: 1) since each path in the query could potentially be either shed or spilled, we need to explore the search space of fusion and its complexity; 2) we need a means to specify and interpret the quality for different substructures to evaluate whether a delayed output of a substructure is satisfactory to the user; 3) fusion candidates may vary in their processing costs and output quality. We need to choose optimal fusion candidates whose corresponding reduced and supplementary queries achieve the highest output quality.

To tackle these challenges, we propose a three-pronged strategy. One, we represent all possible fusion candidates using a Fusion Candidate (FC) lattice. Two, we provide metrics for measuring quality and cost of the alternative reduced queries as well as supplementary queries given some resources. Three, we transform the fusion candidate selection problem into an optimization problem, namely, the design of the fusion candidate that maximizes total output quality.

Contributions. Our contributions are summarized as below:

1. We propose a new calibrated integrated framework for an integrated structure-based shed and spill approach which is able to be applied in any XML stream systems.
2. We formulate our structure-based shedding and spilling problem into an optimization problem, namely, to find a pair of the reduced and supplementary queries that maximizes the output quality based on our structure-based quality and cost model for XML streams.
3. We propose a solution space for fusion candidates which is represented by a Fusion Candidate (FC) lattice. The complexity of FC lattice is $O(3^{f^d})$, where d and f indicate the depth and fan-out of the query pattern tree.
4. We propose an age-based quality model for evaluating the output quality for different reduced and supplementary query pairs.
5. We develop a family of four optimization strategies: OptF, OptSmart, HiX and Fex. OptF and OptSmart are both guaranteed to identify an optimal pair of reduced query and supplementary query, with OptSmart exhibiting significantly less overhead than OptF. HiX and FeX use heuristic-based approaches, which are much more efficient than OptF and OptSmart.
6. Our experimental results demonstrate that our strategies consistently achieve higher quality results compared to the state-of-the-art techniques.

1.5 Dissertation Organization

The rest of this dissertation is organized as follows. The three research topics are discussed in detail in Part I, Part II and Part III in this dissertation respectively. The discussions on each of the three research topics include the relevant research motivation, problem introduction, background, solution description, experimental evaluation and related work respectively. Chapter 24 concludes this dissertation and Chapter 25 describes possible future work.

Part I

Structure-based Shedding for XML Streams

Chapter 2

Preliminaries

2.1 Query Pattern Tree

We support the core subset of XQuery in the form of “for... where... return...” expressions (referred to as FWR) where the “return” clause can contain further FWR expressions; and the “where” clause contains conjunctive selection predicates, each predicate being an operation between a variable and a constant. We assume the queries have been normalized as in [18].

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

Figure 2.1: Query Q1

The example query Q1 in Figure 2.1 is introduced in Chapter 1.4.1. This query returns the item list and contact information including telephone, email and address when they spend more than 100 dollars. The query pattern tree for query Q1 is given in Figure 2.2.

In Figure 2.2, each navigation step in an XPath is mapped to a tree node. We use single line edges to denote the parent-children relationship or attributes and double line edges to denote the ancestor-descendant relationship.

We define the following terms in an XQuery. First, a *context node* corresponds to a context variable in the "FOR" clause, e.g., *$a* in Figure 2.2. Context variables must evaluate to a non-empty set of bindings for the FWR expression to return any result. Second, a pattern that correspond to an XPath in the "RETURN" clause, e.g., $a/*contact*/*tel* or $a//*name*, is called *return pattern* ("r" pattern). Return patterns are optional, meaning even if $a/*contact*/*tel* evaluates to be empty, other elements will still be constructed. Third, a *selection pattern* ("s" pattern) correspond to an XPath in the "WHERE" clause, i.e., it has associated predicates. For instance, the XPath, $*a*/order/*price* in Figure 2.2 is a selection pattern. The "r" and "s" pattern for query Q1 are annotated on their destination elements in Figure 2.2. We call the destination nodes of the return and selection patterns "r" and "s" nodes respectively.

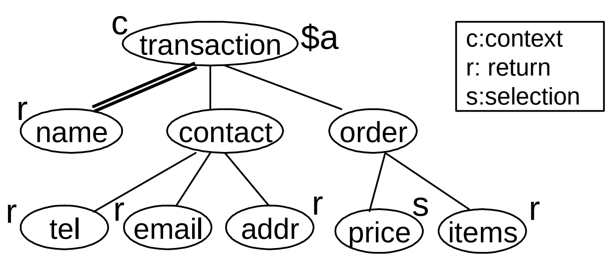


Figure 2.2: Query Pattern Tree for Q1

2.2 Generating Shed Queries

We now investigate how to generate shed queries based on a given query. We distinguish between two terms, sub query and shed query. Sub queries are generated by removing one or multiple nodes from the initial query tree. A shed query is a valid sub query, and it obeys the following rules:

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

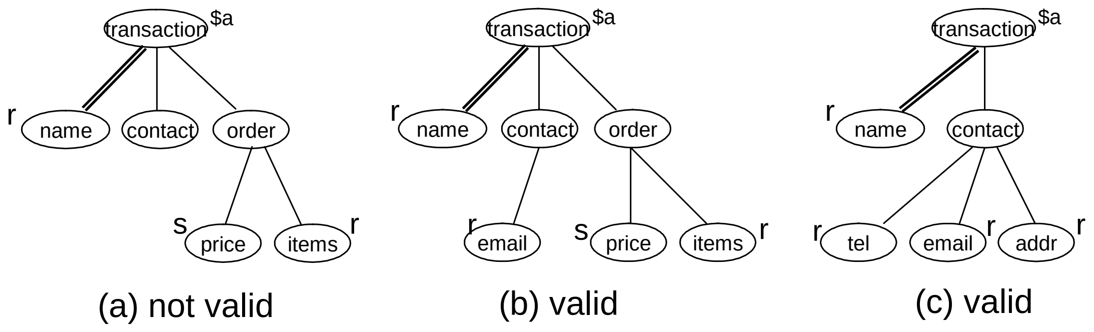


Figure 2.3: Shed Query Trees

For instance, Figure 2.3(a) is not valid because this tree does not need to keep the “contact” element because all children of the “contact” element are removed and the XPath $\$a$/*contact* is neither an “r” nor an “s” pattern. In other words, keeping pattern $\$a$/*contact* in the query does not make any sense since it does not contribute to any returned element or predicate. Figures 2.3(b) and (c) show two valid sub queries for query Q1.

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

Chapter 3

Cost Model

3.1 Automaton Processing Model

As is known, automata are widely used for pattern retrieval over XML token streams [22, 30, 44]. The relevant tokens are assembled into elements to be further filtered or returned as final output elements. The formed elements are then passed up to perform structural join and filtering. An algebra plan located on top of the automaton for query Q1 is shown in Figure 3.1. An Extract operator is responsible for collecting tokens for some pattern and composing them into XML elements. For instance, $Extract\$a//name$ collects tokens to form "name" elements. Structural join operator is responsible for combining the elements from its branch operators based on structural relationship and form a transaction tuple. Observe that the context node $\$a$ in the "FOR" clause is mapped to a structural join. In addition we perform selection on $a/order/price to judge whether the "price" is greater than 100. Thus we have the following query processing tasks in XML stream systems: 1. Using automaton to locate tokens. 2. Extracting tokens. 3. Manipulating buffered data, which includes structural join and selection.

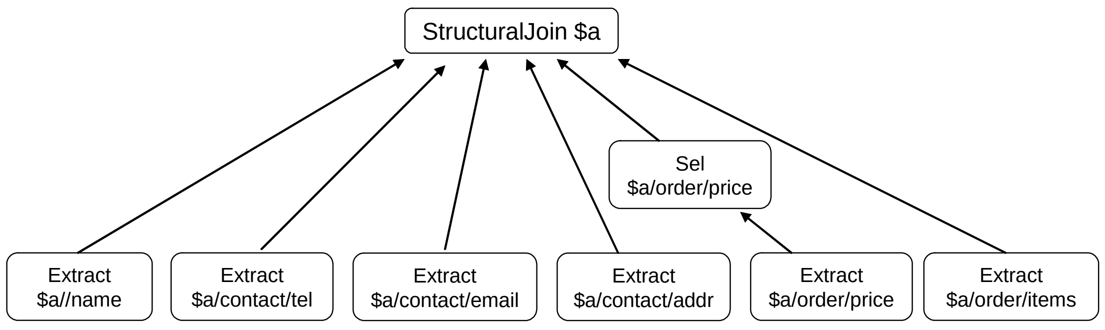


Figure 3.1: An Example Plan

3.2 CPU Cost Model for a Query

We now design a cost model to estimate the processing costs of shed queries for XML streams. This cost model is adapted from the cost model proposed in [57]. In XML streams we measure the query cost for a complete topmost element since it is the basic unit based on which we generate query results. We call the processing time of handling such a topmost element the *Unit Processing Cost* (UPC). For instance, the cost of query Q1 thus is the unit processing cost of handling one “transaction” element.

We divide the UPC for XQuery into three parts: *Unit Locating Cost* (ULC) that measures the processing time spent on automaton retrieval, *Unit Buffering Cost* (UBC) spent on pattern buffering and *Unit Manipulation Cost* (UMC) spent on algebra operations including selection and structural join. UPC is equal to the sum of the cost of these three parts. When we drop either “r” patterns or “s” patterns from the query, we estimate the cost change for these three parts. Note that for a new shed query, its processing cost might not be reduced when dropping “s” pattern. Although it appears that the evaluation cost of the selection pattern is saved, it might need to construct more nodes. In this case the UPC might even be increased if the selectivity of the “s” pattern is not 1. However, due to limited space, we only discuss ULC and UBC here. UMC and the discussion about the

Notation	**Explanation**
N^{P_i}	Number of elements P_i for topmost element.
n_{start}	Total number of start or end tags for a topmost element.
S^{P_i}	Number of tokens contained for a P_i element.
A	Set of states in automaton.
A^{P_i}	Set of states of pattern P_i and its dependent states.
$n_{active}(q)$	the number of times that stack top contains a state q when a start tag arrives
$C_{transit}$	cost of processing a start tag of an element in the query
C_{null}	cost of processing a start tag of an element not in the query
$C_{backtrack}$	cost of popping off states at the stack top
C_{buf}	cost of buffering a token

Table 3.1: Notations Used in Cost Model

selectivity of “s” patterns can be seen in [66].

Unit Locating Cost (ULC). In locating tokens, when an incoming token is a start tag, we need to check whether this start tag will lead to any transitions. If it is transitioned to a new state, tasks to be undertaken may include setting a flag to henceforth buffer tokens or to record the start of a pattern. We call such a transition cost $C_{transit}$. Note that the start tokens of all elements in the query tree will cause such a transition. When there are no states to transition to, an empty state is instead pushed to the stack top. Note that all start tokens of patterns that do not appear in the query tree will lead to such an empty state transition. The cost associated with this case is C_{null}. For instance, when $< id >$ is encountered, an empty state is pushed to the stack top. When the incoming token is an end tag, the automaton pops off the states at the top of the stack. We refer to such popping off cost as $C_{backtrack}$. The popping costs for all end tags are the same. The relevant notations are given in Table 3.1.

We split the ULC into two parts, one considers the cost of locating the start and end tags for elements in the query tree, and the other considers the cost for locating the start and end tags for other elements. The first part can be measured by considering the invo-

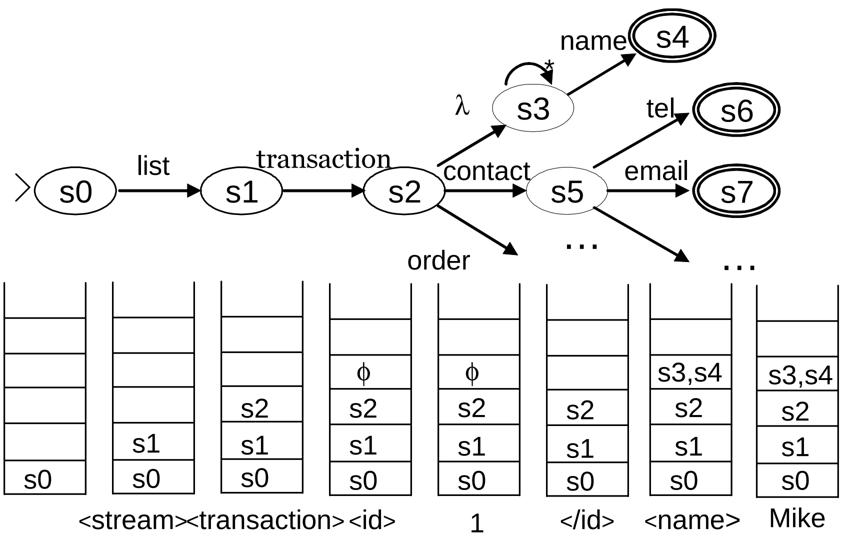


Figure 3.2: Snapshots of Automaton Stack

cation times for each state and the transition cost for a token as below:

$$\sum_{q \in A} n_{active}(q)(C_{transit} + C_{backtrack}) \tag{3.1}$$

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

$$(n_{start} - \sum_{q \in A} n_{active}(q))(C_{null} + C_{backtrack}) \tag{3.2}$$

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

$$\sum\nolimits_{q \in A^{P_i}} n_{active}(q)(C_{transit} + C_{backtrack}) \tag{3.3}$$

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

$$\sum\nolimits_{q \in A^{p_i}} n_{active}(q)(C_{null} + C_{backtrack}) \tag{3.4}$$

Thus Eq(3.3)- Eq(3.4) indicate the savings in locating costs gained by switching from the initial query to this shed query Q_s.

Unit Buffering Cost (UBC). In our query engine, we only store those tokens that are required for the further processing of the query. As we mentioned, the Extract operators are responsible for buffering those tokens. Thus each “r” and “s” pattern has a corresponding Extract operator. Such buffering cost for a topmost element is defined as UBC (Unit Buffering Cost). Extract operators are invoked when the corresponding states are reached in the automaton. For example, in Figure 3.2, state $s4$ would invoke an Extract operator to store the whole “name” element. In addition we assume here the buffering cost is the same for all individual tokens.

Our buffer manager uses pointers to refer to elements. Thus we do not store the same token more than once. Three query examples are shown in Figure 3.3. In Figure 3.3(a) and 3.3(b), the parent pattern and its children patterns overlap. Since both the parent and the children are to be returned, we only need to store the parent pattern $p1$ and set a reference for its children $p2$, $p3$ and $p4$ pointing to $p1$. In this case, the buffering cost is equal to the buffering cost of the parent pattern $p1$. However, in Figure 3.3(c), since the parent is not an “r” pattern, only its children are to be returned. The buffering cost is

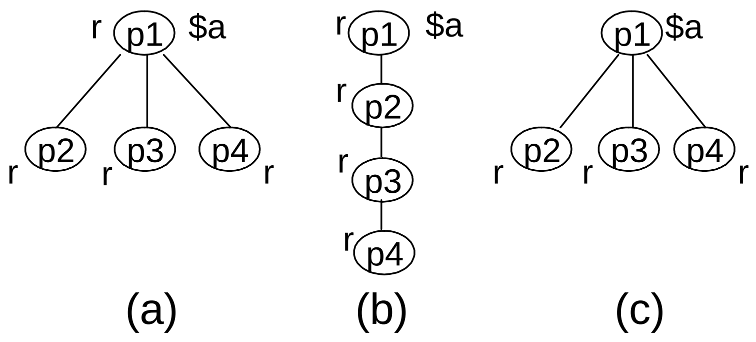


Figure 3.3: Buffer Sharing Examples

equal to the buffering cost of all the children. Hence, for a given query, we need to find all non-overlapping topmost patterns which are either “r” patterns or “s” patterns, called henceforth the *storing pattern set*. The storing pattern set can be obtained by traversing the query tree in a breadth-first manner [66].

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

$$UBC(Q) \; = \; \sum_{p \in R} N^p S^p C_{buf} \tag{3.5}$$

Runtime Statistics Collection. We collect the statistics needed for the costing using the estimation parameters described above. We piggyback statistics gathering as part of query execution. For instance, we attach counters to automaton states to calculate N^{P_i}, n_{start} and $n_{active}(q)$. And we collect s^{P_i} in Extract operators. We then use these statistics to estimate the cost of shed queries using the formulas given above. Note that some cost parameters in Table 3.1 such as $C_{transit}$, C_{null} and C_{buf} are constants. We do not need to measure them during the query execution.

Chapter 4

Preference Model for Queries

Value-based Preferences vs. Structure-based Preferences. In many practical applications, some output results are considered more important than other output tuples. For instance, the user might be interested in red cars when buying new cars. In this case the utility of the tuple whose color attribute is equal to "red" is higher than those of the tuples whose colors are not "red." Aurora first considered such value-based preference as part of the QoS requirement and proposed semantic load shedding techniques [59] to maximize output utility. In this case, semantic load shedding is achieved by adopting a value-based filter. We can easily incorporate such value-based preferences and their filter-based shedding approach in the XML stream scenario. However, this is not our main interest. Instead, we are interested in exploring the structure-based preference in XML stream processing. In the XML stream scenario, the input stream as well as the output result are composed of different XML subelements, and hence more complex than relational tuples. The importance of different elements in an XML tree may vary due to their semantics. As illustrated in Chapter 1.4.1, in query Q1, the "email" element is considered more important than the "addr" element as "email" is a faster and more convenient means to contact customers.

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

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

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

We support two alternative types of preference specification on query patterns. One uses prioritized preference [36] to qualitatively express the relative ranking among different patterns. The other uses a quantitative approach [26, 27] that directly scores the importance of the patterns. Users are free to choose either the Numerical Preference Model (NPM) or the Prioritized Preference Model (PPM) to represent their preferences on query patterns. For preferences specified by PPM, we translate the prioritized preferences to numerical forms using a score formula. Note that in both cases we use the quantitative metric to compute the utilities for the shed queries.

4.1 Numerical Preference Model (NPM)

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

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

Here $\nu(P_j)$ is a constant value between [0,1]. An example of utility assignment for query Q1 is shown in Figure 4.1 (the utility is labeled on the destination node of each pattern).

4.2 Prioritized Preference Model (PPM)

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

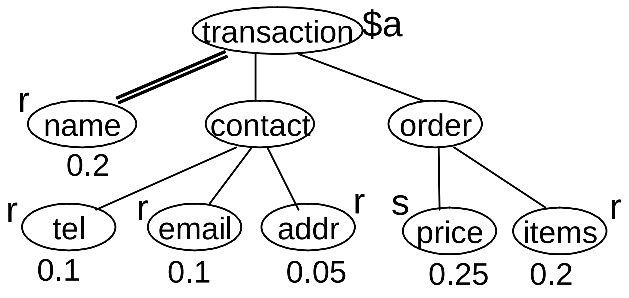


Figure 4.1: Query Tree with Preference for Q1

patterns. An example prioritized preference for query Q1 is:

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

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

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

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

4.3 Scoring Scheme for Patterns without Preferences

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

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

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

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

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

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

4.4 Computing Utilities for Queries

After the quantitative preferences for all the patterns in the query are determined, we can calculate the utility of the original query and the shed queries derived from the original query. If a pattern appears in a query tree of a shed query, that means it will be considered in the query and its utility is obtained. We use the utility of a query to indicate the amount of utility users gain by executing this particular query Q on a single topmost element, in other words, how much utility is obtained by including all the patterns in this shed query.

It can be calculated as

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

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

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

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

An extension of XPath is proposed in [37] that incorporates value-based preferences into XPath. Similarly we can easily extend the XQuery syntax to integrate our structure-based preferences into an XQuery expression as below:

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

Chapter 5

Shedding Algorithms

5.1 Decide When to Shed

The problem of deciding when the system needs to shed input data has been discussed in other works [59]. This is not specific to XML stream systems. In our system we adopt the following approach for simplicity. We assume a fixed memory to buffer the incoming XML stream data. As soon as all tokens in an XML element have been processed, we clean those tokens from the buffer. We assume a threshold on the memory buffer that allows us to endure periodic spikes of the input without causing any overflow. During execution, we monitor the current memory buffer. When buffer occupancy exceeds the threshold, we trigger the shedding algorithm.

5.2 Formulation of Shedding Problem

Let us assume that the shed query set is $\{Q_0, Q_1, ..Q_n\}$ where Q_0 is the empty query and Q_1 is the original query. Here empty query just drops all the tokens of a topmost element. The reason why we introduce empty query Q_0 into shed query set is for the convenience of the formalization of the shedding problem, so that all the input elements are consumed by shed queries. Since this empty query does not generate any output, we assume the utility of empty query Q_0 denoted by ν_0 and the UPC of Q_0 denoted by C_0 are both zero. The goal of the shedding problem is to find which shed queries will be chosen to run in order to achieve maximum utility. We have the following inputs to our shedding problem.

1. Data arrival rate λ in the unit of topmost elements per time unit.

2. Utilities of candidates in the query set $\{\nu_0, \nu_1, ..\nu_n\}$.

3. Processing costs (in time units) of queries in the set $\{C_0, C_1, ..C_n\}$.

4. The number of time units for shedding query to execute, C, denoting the available CPU resources.

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

Another option is to restrict the number of shed queries to two. As mentioned before, there might not exist such a shed query from the query set whose processing speed is as fast as input arrival rate except empty query. It implies that if picking two queries from the shed queries and none of them is the empty query, we cannot handle all input data.

Thus picking the empty query is necessary. Given that the empty query cost is zero, we can formulate this problem below:

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

We want to maximize output utility $x_i * v_i$. The number of elements to drop (corresponding to empty query) is thus equal to $\lambda - x_i$. Note that the current state-of-the-art shedding techniques [11, 59] can be regarded as a special case for allowing two shed queries, as they typically pick the original query and empty query.

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

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

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

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

$$X(s) \; = \; \sum_{i=0}^{n} x_i \tag{5.1}$$

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

$$C(s) \; = \sum_{i=0}^{n} x_i * C_i \tag{5.2}$$

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

$$\sum_{i=0}^{n} x_i \nu_i \tag{5.3}$$

Subject to

$$X(s) \; = C * \lambda$$

$$\texttt{and} \;\; C(s) \; \leq \; C \tag{5.4}$$

Note that the cost of all shed queries are measured in time units, thus they are all non-negative integers. We thus conclude that this problem is an instance of the knapsack problem [35]. We propose two solutions for this problem as described below.

5.3 OptShed Approach

OptShed uses a dynamic programming solution [53]. To state our approach, we construct a matrix of sub-problems:

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

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

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

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

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

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

From the matrix of sub-problems, we can see that we need to repeat the calculation of $\phi(\tilde{c})$ nC times to get the final result, and each calculation can be finished using a max-value searching algorithm, whose time cost is O($\log_2 C$) [53]. Thus the total time complexity is O($nC \log_2 C$).

5.4 FastShed Approach

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

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

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

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

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

The details are described in Algorithm 1.

Algorithm 1 FastShed

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

In FastShed, the ratio sorting cost is O($n \log n$) and cost of “for” loop is O(n) respectively. So the total time complexity is O($n \log n$). Normally, $n \ll C$, so FastShed is much faster than OptShed, though FastShed cannot guarantee to find an optimal solution. However, in Chapter 7, the experimental results show that FastShed indeed tends to find a solution very close to the optimal solution for most cases.

Chapter 6

Shedding Mechanism Implementation

In this chapter, we examine the implementation of different shedding approaches in XML stream systems. For relational stream systems, one common implementation is to insert drop boxes into the plan [7, 11, 59]. However, many XML stream systems use automata to recognize relevant elements on incoming token streams. In this case, we can consider at least two options where the input data can be dropped. One place is when we recognize the tokens using automaton, the other place is after we have form the elements from extracted tokens. Since dropping them as early as possible can avoid wasted work, we propose to push the shedding directly into the automaton as described below.

6.1 In-Automata Shedding Mechanism

Here we propose to incorporate shedding into the automaton by disabling states. Assume we want to drop patterns $a//name and $a/contact/tel. Figure 6.1 shows where to insert drop boxes in the automaton. To drop pattern $a//name, the automaton would temporarily remove the transition from state $s2$ to $s3$. When the start tag of $name$ element arrives, state $s3$ and $s4$ are not reachable. Thus it would not invoke its downstream operator,

$Extract\$a//name$. $Extract\$a//name$ will then be labeled with a "dropped" flag. This flag guarantees that the downstream $StructuralJoin\$a$ operator works correctly. Thus when $StructuralJoin\$a$ checks its input operators one by one, if an input operator is labeled with a "dropped" flag, $StructuralJoin\$a$ skips this input.

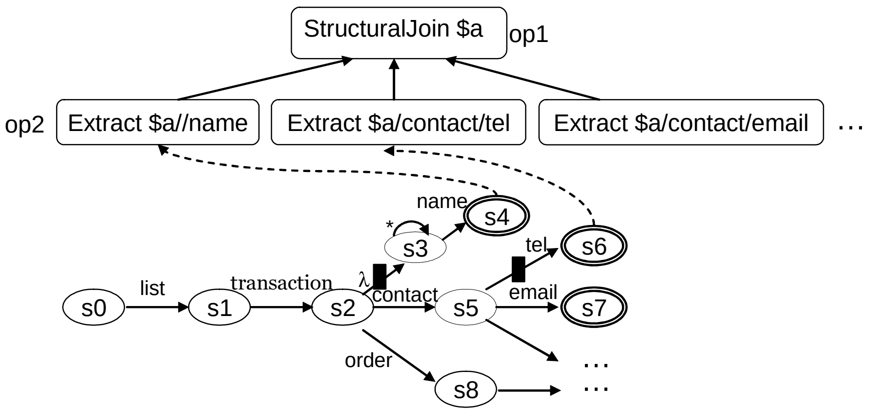


Figure 6.1: Disable Transition Strategy

6.2 Random Shedding in XML Streams

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

6.3 Shed Query Switching at Run-time

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

Chapter 7

Experimental Results

We used ToXgene [12] to generate XML documents as our testing data. All experiments were run on a 2.8GHz Pentium processor with 512MB memory. We used query Q1 as testing query and the testing data files are about 30 MB. We performed four sets of experiments. The first one shows that output utility changes with varying arrival rates for all three shedding approaches (Random, OptShed and FastShed). The second set of experiments demonstrates that different distributions of pattern preference settings and pattern sizes impact the output utility. The third set compares the overhead of three shedding strategies. It shows that FastShed has little overhead, similar to Random shedding. However, the overhead of OptShed becomes big for large query sizes. The final set of experiments shows FastShed achieves close-to-maximum utility in practically all cases considered.

7.1 Comparison Among Three Shedding Approaches

In this set of experiments, we studied the output utility changes with varying arrival rates for the three shedding approaches. Fig. 7.1 shows the output data utility per second for query Q1. Note that in Fig. 7.1 the three slopes increased the same way when arrival rate is less than 180 topmost elements/s because no shedding happens at that time. After the arrival rate reaches 180 topmost elements/s, the utility of Random remained stable because it has reached its processing capacity. However, FastShed and OptShed achieved higher utility because they chose a shed query which generates higher utility than the Random approach.

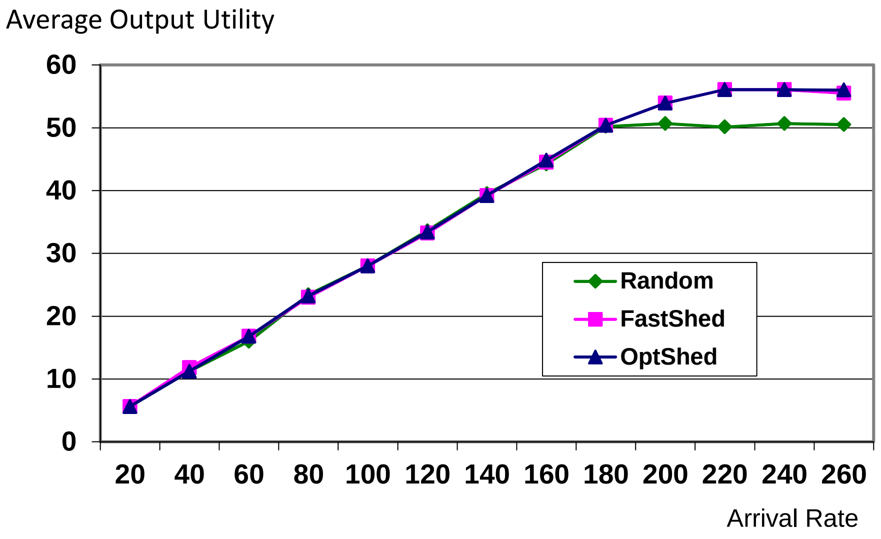


Figure 7.1: Output Utility Changes with Varying Arrival Rates

7.2 Effect of Preference and Pattern Size

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

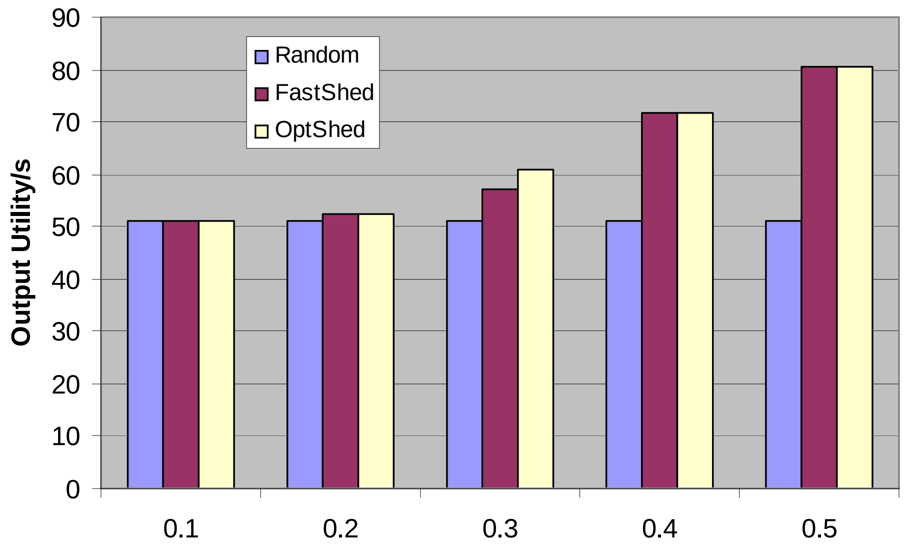


Figure 7.2: Data Utilities for Varying Preference Assignments

We used five different sets of preference settings which differ in their standard deviations. We run query Q1 on the same data set. Each pattern had the same size and each set has the same utility for the initial query. Figure 7.2 shows that the output utility is higher when there is a bigger variance among pattern preference settings for FastShed and OptShed. We observe that the utilities of the query achieved by the Random approach are the same because the initial query is executed in this case. However, OptShed and FastShed performed differently when the standard deviation for preferences changes. Observe that when the standard deviation of preference values was small, there is little difference among utilities for the three approaches. However, the difference of output utility was significant when the standard deviation of preference values reaches 0.5.

To illustrate the output utility is affected by the pattern sizes, we generated five testing data files which differed in their standard deviation of element size. We ran the query Q2 below.

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

Note that each data file only contained the elements in the query and the sums of all element sizes in each data file were all equal to 200 tokens. In addition we assume all patterns in the query are independent and of equal preference. Figure 7.3 shows the output utility changes with varying standard deviation of pattern size during the same time period. Observe that for the Random approach, the output utilities did not change a lot since the UPC of the original query for these four data files are almost the same. However, for FastShed and OptShed, the output utility was much higher than the utilities achieved by Random approach when the standard deviation of pattern size increased. This is because the shed queries with smaller patterns has smaller locating cost and buffering cost, resulting in lower overall processing cost. In this case, FastShed and OptShed would pick such shed queries since they have relatively higher utility/cost ratios and thus higher utilities.

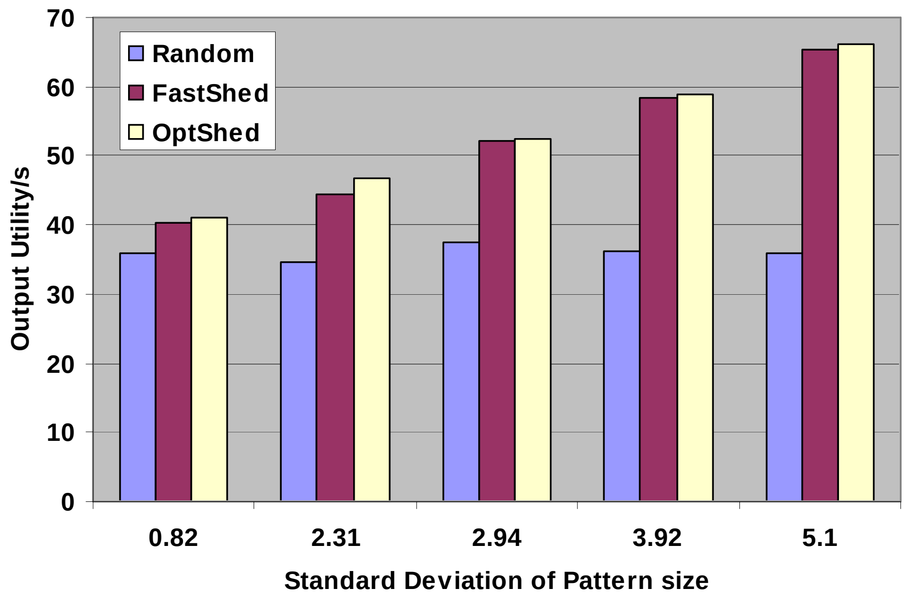


Figure 7.3: Data Utilities for Varying Pattern Size

7.3 Overhead of Shedding Approaches

Here we studied the overhead of the three shedding strategies. The overhead was measured by the time spent on choosing which shed query to run during the shedding phase. We studied whether with more complex query the overhead increases dramatically. We used five queries which vary in the number of patterns. From Figure 7.4, we observe even when the query became complex, the overhead of FastShed was still very small, although it was a bit higher than Random shedding. But it did not scale when the query became more complex. However, for OptShed, overhead was already very high when the number of patterns in the query is 5. Thus the overhead of OptShed is very big, implying it as an undesirable choice.

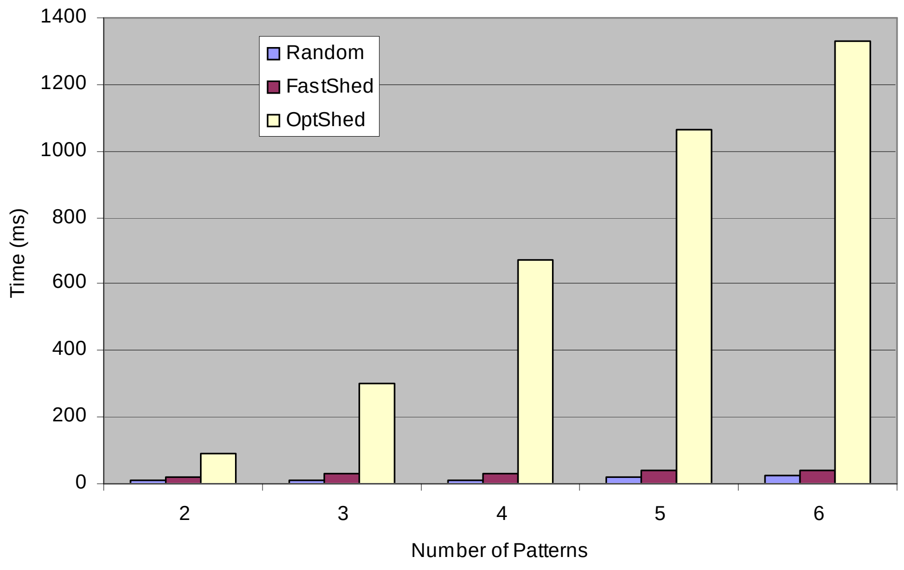


Figure 7.4: Overhead of Three Shedding Approaches

7.4 Additional Experiments on Three Approaches

In the first two experiments above, we observed that FastShed and OptShed performed better than Random shedding on output utility. However, we only compared them based on a limited number of preference settings. Now, we want to study performance of these methods over a wide range of cases. We generated 1000 sets of sample costs and utility measures, where a sample set is generated by assigning preferences to different query patterns randomly. The costs of different shed queries in a sample set were assigned randomly in the range [10, 20], and at the same time ensuring that the cost of a "smaller" query was less than the cost of a "bigger" query. Then we ran the three approaches on these 1000 sets of sample data and compared their output utility. Figure 7.5(a) shows the histogram on the utility ratios of FastShed over OptShed. We observe that these ratios are skewed to the left. About 80% of them are over 0.8. This means that FastShed can get close to optimal results in most cases. Figure 7.5(b) shows the histogram of output utility ratios of Random over FastShed. Observe that these ratios were skewed to the right. Most of them are less than 0.6. Thus FastShed is much better than Random shedding.

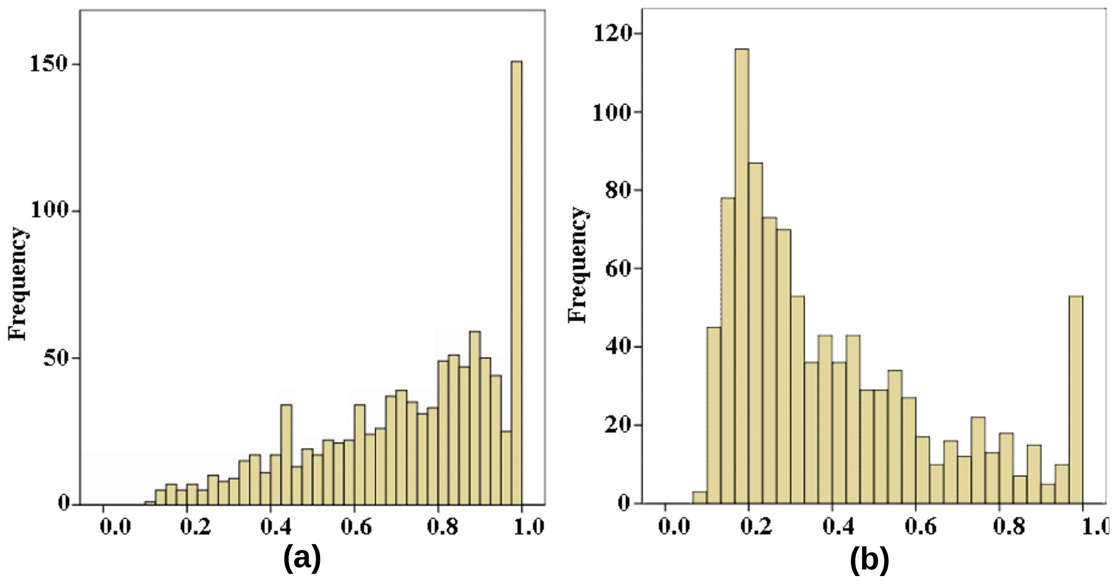


Figure 7.5: Utility Ratios of (a) FastShed over OptShed (b) Random over FastShed

Chapter 8

Related Work

In streaming systems, approximate query processing has been considered an effective method for trading off performance with accuracy [21, 49, 56, 59, 61]. However, most approximate query processing work has been focused on relational streams. Load shedding and sampling data are two most common ways to reduce system workload. Load shedding on streaming data has firstly been proposed in the Aurora system [59]. This work introduces two types of load shedding: random and semantic load shedding. Based on the analysis of the loss/gain rate, the random load shedding strategy will determine the amount to shed to guarantee the output rate. For semantic drop, they assume that different tuple values may vary in term of utility to application. In this case, maximizing the utility of output data is their goal. We have the same goal of maximizing the output data utility in XML streams. However, instead of a simplistic model of certain domain value denoting utility, we consider the complexity as well as importance of XML result structures in order to make shed query decisions.

Most approximate query processing works focus on the max-subset goal, which is, to maximize the output rate [7, 21, 28]. [21] provides an optimal offline algorithm for join processing with sliding windows where the tuples that will arrive in the future are known to the algorithm. An online algorithm that does not know which tuples will arrive in the future is given under assumption about certain arrival possibilities. [56] proposes a novel age-based stream model and describes the load shedding approach for join processing with sliding windows under limited memory resources. We could apply their techniques into join processing among multiple XML stream systems if our goal is to get max-subset instead of maximizing output utility. In addition, we explore how to choose shed queries to maximize output utility for XML streams under limited CPU resources. [28] provides an adaptive CPU load shedding approach for window stream joins in relational stream systems. It follows a selective processing methodology by keeping tuples within the windows, but processing them against a subset of the tuples in the opposite window. We cannot apply these approximate processing techniques directly into our work since we are targeting a single XML stream without window constraints.

[7] investigates the approach to do load shedding for sliding windows on conjunctive queries. The goal is to choose the plan with drop boxes inserted that maximize the output rate of the partial answer query. It addresses two problems, one is the optimal placement of the drop boxes in an execution plan and the optimal setting of the sampling rate. The second is the choice of the plan to shed load from. This work combines the problem of finding an optimal execution plan and exploring the strategy on the placement of the drop boxes into a single optimization problem. Their approach is orthogonal to our approach. Some works reduce the workload by changing the query explicitly. [49] changes the query at the operator level. This is similar to our removal of some patterns from the query. However, we consider the complexity of XML result structures.

Preference model is a natural way for decision making purpose. It is used in many applications, including e-commerce and personalized web services. As mentioned before, Aurora [59] combines the utility of different tuple values into quality of service metric. [38] proposes Preference SQL, an extension language SQL which is able to support user-definable preference for personalized search engines. It supports some basic preference types, like approximation, maximization and favorites preference, as well as complex preference. Preference XPath [37] provides a language to help users in E-commerce to express explicit preference in the form of XPath query. For view synchronization in dynamic distributed environments, EVE [40] proposes E-SQL, an extended view definition language by which view definer can embed their preferences about view evolution into the view definition.

Part II

Structure-based Spilling for XML Streams

Chapter 9

Overview of Structure-based Spilling Approach

The architecture of our spilling framework is shown in Figure 9.1. After the queries are registered with the query engine, an initial plan is generated and optimized. The execution engine will instantiate the query plan and start processing input streams. The problem of deciding when the system needs to spill data is not a question specific to XML streams. Any existing approach from the literature [48, 63] could be employed here. We employ a memory buffer to store input stream data. As soon as a token is processed, we clean this token from the buffer. We assume a threshold on the memory buffer that allows us to endure periodic spikes of the input. When buffer occupancy exceeds the given threshold, we trigger the spilling.

When spilling is triggered, first, the possible spilling candidates are examined. We then derive the reduced queries for each spilling candidate. The query optimizer runs the optimization algorithm to pick the optimal reduced query. Finally the reduced query is instantiated, in place of the previously active query, initiating the spilling process. Later when the arrival speed becomes near zero, we invoke the clean up processing to generate supplementary results based on disk-resident data.

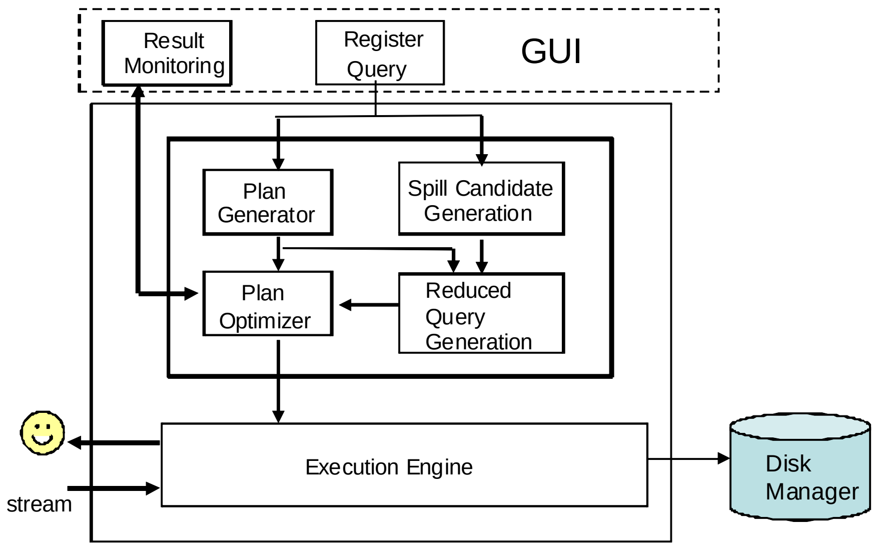
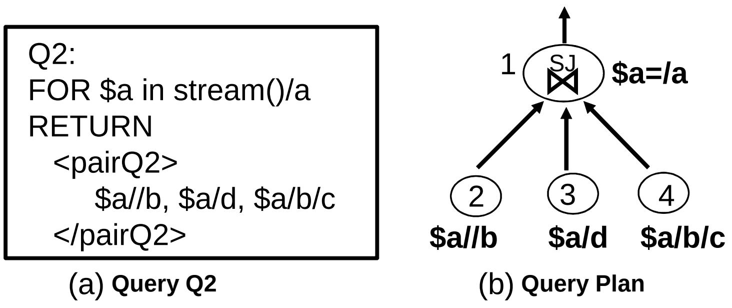


Figure 9.1: Architecture for Spilling Framework



```
Q2:
FOR $a in stream()/a
RETURN
  <pairQ2>
     $a//b, $a/d, $a/b/c
  </pairQ2>
```

Figure 9.2: Query Q2 and Its Plan

Recall that any path and any number of paths in the query can be spilled. We describe the details of possible spilling candidates in Chapter 11. Let us use query Q2 introduced in Chapter 1.4.2 as our example (query Q2 and its plan are shown in Figure 9.2). Now let us illustrate how to pick the optimal spilling candidate to produce maximum output quality. We require the optimal reduced query should be able to consume all the input, i.e., the processing speed of the optimal reduced query should be faster than or equal to the arrival rate. For example, assume we have two spilling candidates for Q2, $/a//b$ and $/a/b/c$.

The data is shown in Figure 9.3(a). Figures 9.3(b) and (c) list output results after spilling $/a//b$ and $/a/b/c$ respectively. Assume the arrival rate is 500 topmost elements/sec (for Q2, a is the topmost element). Assume the cost to produce each <pairQ2> element when spilling $/a//b$ is 0.6 milliseconds. The cost of producing each <pairQ2> when spilling $/a/b/c$ is 1 millisecond. The processing rates when spilling $/a//b$ and $/a/b/c$ are 1000/0.6 =1333 and 1000/1=1000 respectively. Both values are greater than the arrival rate. Therefore spilling either $/a//b$ or $/a/b/c$ can both meet our goal of consuming all the input. However, the output quality for each spilling path is different. When spilling $/a//b$, since only d elements are present in the results, the quality for each <pairQ2> is 1 (quality computation is detailed in Chapter 13). The quality when spilling $/a/b/c$ is 3 since b (including partial b and complete b) and d elements are returned. In this case, the output quality when spilling $/a/b/c$ within 1 second is 500 * 3 and the quality when spilling $/a//b$ is 500 * 1. Therefore spilling path $/a/b/c$ yields higher output quality than $/a//b$. We will describe the detailed algorithm to find an optimal candidate in Chapter 15. This structural spilling framework is general and can be applied in any XML stream engine. The detailed explanation of why our spilling framework is general is explained later in this chapter.

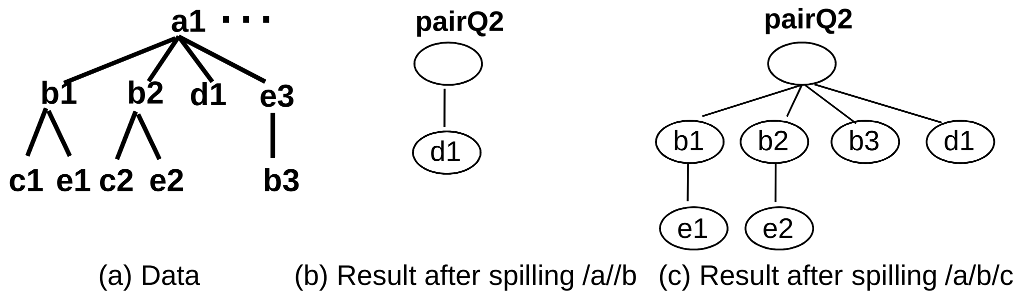

a1 · · ·
b1
b2
d1
e3
c1
e1
c2
e2
b3
pairQ2
d1
pairQ2
b1
b2
b3
d1
e1
e2
(a) Data
(b) Result after spilling /a//b
(c) Result after spilling /a/b/c

Figure 9.3: Data and Output for Q2

To eventually produce the entire, yet duplicate-free result set, we have to generate supplementary results correctly. We propose a complementary output model, which extends from the hole-filler model in [25], to facilitate the matching of the supplementary results with prior generated output. In addition, we examine what extra data must be flushed to guarantee the generation of the correct “delta” structure in supplementary results. The details of generating supplementary results can be found in Chapter 14.

General Framework for Structural Spilling. The framework we propose to use to address the structural spilling problem described in this work is general, meaning it could be applied to any XML stream management system. Recall that to solve the structural spilling problem, we have to examine the possible spilling candidates, derive the spilling effects, measure the quality as well as cost of the reduced queries, and run the optimization algorithm to choose the optimal reduced query. The spill candidates are generated based on the query pattern tree, which is directly derived from the query. For each spilling candidate, determining the spilling effects in the query is resolved by deciding the data dependency relationship between the spilled path and paths in the query. Hence determining spilling effects is related to the query semantics. It is not related to the specifics of the implementation of query processing. The quality model in Chapter 13 measures the output quality based on the query result. Again this is solely based on the query semantics and thus, general. Note that our optimization algorithms to search the optimal reduced query are cost-based approaches. Obviously, the execution cost measurement for each spilling candidate in other stream engines may be different from that of our system because of the specifics of query processing. For this, we can plug in the cost model of other stream engines. In this case, the optimality of our search algorithms can still be guaranteed.

Chapter 10

Background

Queries Supported. We support a subset of XQuery in this work. Basically, we allow (1) “for... where... return...” expressions (referred to as FWR) where the “return” clause can further contain FWR expressions; and (2) conjunctive selection predicates where each predicate is an operation between a variable and a constant. The grammar of the supported XQuery expressions is shown in Figure 10.1. A large range of common XQueries can be rewritten into this subset [47]. A query with “let” clauses can be rewritten into an XQuery without “let” clauses (by Rule NR1 in [47]). A query with FWR expressions nested within a “for” clause can also be rewritten into our supported subset format (by Rule NR_4 in [47]). The filter expression in an XPath can be moved into the “where” clause.

Algebraic Query Processing. We assume the queries have been normalized using the techniques in [18]. Queries are then translated into a plan. Namely, for each binding variable in the “for” clause, a structural join is conducted between the binding variable and the paths in the “return” clause. Paths in the “return” clause are translated into inputs to the structural join operator. The expressions in the “where” clause are mapped to select operators. Finally a tagging function is on top of the plan taking care of the element

```
CoreExpr ::= ForClause WhereClause? ReturnClause
            | PathExpr
PathExpr ::= PathExpr "/"|"//" TagName|"*"
            | varName
            | streamName
ForClause ::= "for" "$"varName "in" PathExpr
            ("," "$"varName "in" PathExpr)*
WhereClause :: = "where" BooleanExpr
BooleanExpr ::= PathExpr CompareExpr Constant
            | BooleanExpr and BooleanExpr
            | PathExpr
CompareExpr ::= " ="|"! ="|" <"|" <="|" >"|" >="
ReturnClause = "return" CoreExpr
            |<tagName>CoreExpr ("," CoreExpr)* </tagName>
```

Figure 10.1: Grammar of Supported XQuery Subset

construction. Here we focus primarily on the structural join, the core part of the XQuery plan, while tagging is not further discussed. For instance, for the plan in Figure 1.3, structural join is conducted between $\$a$ and each of its branches.

Basic Processing Unit (BPU) refers to the smallest input data unit based on which we can produce results independently. It can be a document or a topmost element extracted by the query. When we encounter the end of a BPU in the incoming data, we can produce the result structure. For example, for query Q2, the BPU is an a element on path $/a$. When </a> is encountered, we can produce <pairQ2> result structures. This provides an efficient way to produce output as early as possible for XML streams [30]. In this work, BPU is the topmost element in the query tree.

Chapter 11

Spill Candidate Space

In this chapter we examine all possible spill candidates. To do this, we represent the query using a query pattern tree. For example, the query pattern tree for Q2 is given in Figure 11.2(a). Each node in the query tree indicates an XPath expression. The semantics of the supported XPath expression can be found in Chapter 10. We use single line edges to denote the parent-children relationship and double line edges to denote the ancestor-descendant relationship.

We assume any node and any number of nodes in the query tree can be spilled. Each of them forms a spill candidate. To analyze the total number of potential spilling candidates, consider a complete query pattern tree with depth d and fixed fan-out f. The total number of nodes in the query tree $|T|$ = $\sum_{i=1}^{d-1} f^i = \frac{f^d-1}{f-1}$. Since any number of nodes in the query tree can be spilled, the total number of potential spilling candidates is $C_{|T|}^0 + C_{|T|}^1 + ... + C_{|T|}^{|T|} = 2^{|T|}$, which is bounded by $O(2^{f^d})$.

An example query tree and its possible candidates are shown in Figure 11.1. Query tree is shown on the left and its possible candidates are shown on the right. Each node in the lattice represents one candidate. The top candidate means spilling nothing (i.e., initial query). The bottom candidate indicates spilling everything (i.e., empty query). Each level

i lists all candidates spilling i nodes from query tree. The candidate space scales quickly since it is exponential in the number of nodes in the query tree.

We now reduce the spill candidate space using the insight that some candidates may result in the same spilling effects. Recall that when we spill data corresponding to a path p from the query tree, all its descendants are also flushed to disk. This leads to the following observation:

Observation 11.0.1. *If a spill candidate includes two nodes that satisfy the ancestor-descendant (or parent-child) relationship, it has the same spilling effect as the candidate containing the ancestor (parent respectively) node.*

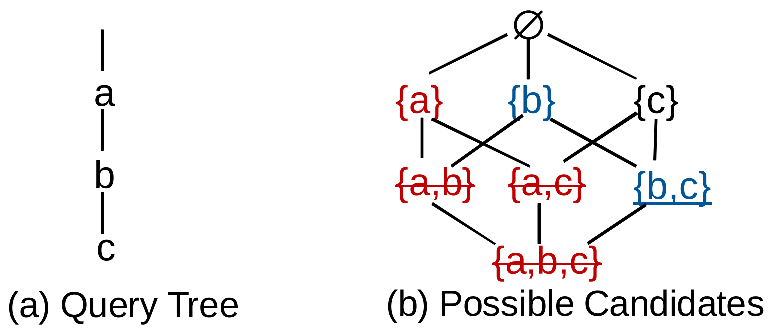


Figure 11.1: Query Tree and Its Spill Candidates

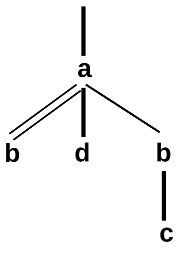
For instance, in Figure 11.1(b), the underlined candidate $\{b, c\}$ has the same spilling effect as $\{b\}$. The candidates with strike-through have the same spilling effect as $\{a\}$. Clearly, we should avoid examining such candidates with the same spilling effects. Hence we introduce a minimum non-redundant spill candidate space.

Minimum Candidate Space. We design an algorithm that generates the minimum set of all non-redundant spill candidates. The idea is to generate non-redundant candidates from the subtrees recursively. For a tree of height h, to generate all possible non-redundant candidates, it picks zero or one candidate from the set of candidates generated by each subtree of height $h - 1$ and composes them to one new candidate. Or, it can also generate a new candidate which consists of a single root node. The algorithm that generates the minimum set of all non-redundant spill candidates is described below:

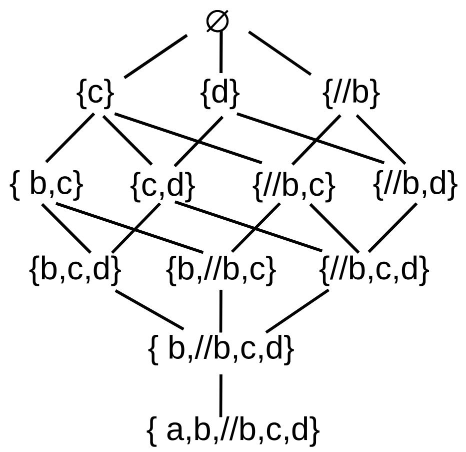
Algorithm 2 minCandidates

```
Input: Query Tree $T$
Output: candidate set $S$
void minCandidates(Node root)
if root is leaf then
   return $\{root\}$;
else
   for each child $C_i$ do
      $S_i$ = minCandidates($C_i$);
      $S_i$ = $S_i \cup \{\emptyset\}$;
   end for
   //Assume root has w children. Generate candidates.
   $S$ = $S_1 \times S_2 ... \times S_w$;
   $S$ = $S \cup \{root\}$;
   return $S$;
end if
```

The total number of potential spilling candidates generated using this algorithm is $O(2^{fd})$. The minimum spill candidate space for query Q2 is shown in Figure 11.2(b). Its size is much smaller than that of the original candidate space which is 2^5 = 32.



(a) Query Tree for Q2



(b) Minimum Spill Candidate Space

Figure 11.2: Minimum Candidate Space for Q2

Chapter 12

Generate Correct Reduced Output

12.1 Determine Spilling Effects

For each spill candidate, we need to derive its corresponding reduced query and generate the correct reduced output. As shown in chapter 1.4.2, when a path is spilled, multiple paths in the query may be affected. To generate the reduced output correctly, we have to determine the spilling effects on the paths in “for”, “where” and “return” clauses for each spilling candidate. Each path in the query corresponds to a set of subtrees in the document. For instance, $/a/b$ returns the subtrees rooted at nodes b whose parents are of type a. Due to spilling, either the root or the non-root nodes in the subtree can be missing. Here we define two categories of spilling effects on paths in the query to distinguish between different missing locations of the subtrees:

- **Root missing or unaffected**. When the roots of subtrees for a query path are missing, we call this *root missing*. Otherwise, it is *unaffected*. For instance, for path $/a//b$, the roots of some subtrees are missing when spilling $/a/b$. This is because path $/a/b$ is contained by $/a//b$. In other words, they satisfy the following relationship:

$$P \bigcap S//* \neq \emptyset \tag{12.1}$$

Here P indicates a path in the query and S indicates the spilled path.

- **Subpart missing or unaffected**. When non-root nodes in the subtrees corresponding to a path in the query are missing, we call it *subpart missing*. Otherwise, it is unaffected. For instance, $/a/b$ is subpart missing when spilling $/a/b/c$ because c nodes in the subtrees are missing due to spilling. The query paths which are subpart missing satisfy the following relationship:

$$P/*//* \bigcap S//* \neq \emptyset \tag{12.2}$$

To determine root missing and subpart missing, we use the approach in [46] which constructs the product automaton of P and S. The complexity of this approach is O(|P|*|S|). Since these two categories are orthogonal, there are 2*2=4 combinations. They are:

- *Root missing and subpart missing* (SRAM). E.g., when spilling $/a//b$, $/a/b$ is SRAM because both root and subpart are missing.
- *Root unaffected and subpart missing* (SAM). E.g., $/a/b$ is SAM when spilling $/a/b/c$ since c nodes in subtrees are missing.
- *Root missing and subpart unaffected* (RAM). This is not possible. Because we assume when a path is spilled, all its descendants are also spilled.
- *Root unaffected and subpart unaffected* (UA). In this case, both root and subpart are unaffected.

12.2 Reduced Query Execution

We now describe how to execute a reduced query based on the knowledge of spilling effects. The reduced query results are output as long as the result is correct, even if the result structures are partial. In other words, the reduced query execution should satisfy the maximal output property [54]. Therefore we propose the following policies for reduced query execution so that we can produce as much correct output as possible.

- **Affected path in "for" clause**. When the binding variable is SRAM, the number of bindings may be reduced. In this case we can still produce output as long as the binding variable does not return empty. When the binding variable is subpart missing (SAM), although a subpart of the binding variable is missing, it does not affect the number of iterations of the "loop counter". Therefore SAM on the "for" path does not affect result generation.

  Example 12.2.1. *Figure 12.1(a) shows the case when the binding variable is SAM. In Figure 12.1(a), the spilled path is $/a/b$. The binding variable $\$a$ is SAM due to spilling $/a/b$. The iterations of "for" loop are unaffected.*

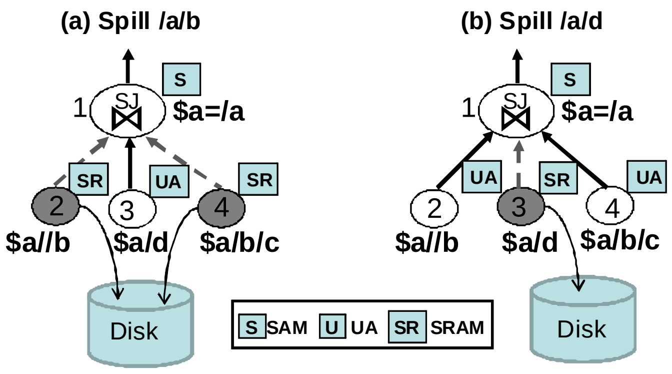


Figure 12.1: Plan for Q2 with Spilling Effects

- **Affected path in "return" clause**. The structural join is conducted between a binding variable V and all its branches. Based on query semantics, the structural join between a binding variable V and one branch $B(i)$ is independent from the structural join between V and other branches. Therefore a "return" path being affected by spilling does not block the output of other "return" paths in the same FWR block.

Example 12.2.2. *Figure 12.1(a) shows the case that the returned paths* $\$a//b$ *and* $\$a/b/c$ *are both SRAM due to spilling* $/a/b$*. For data in Figure 9.3(a), only* $b3$ *and* $d1$ *are present in the* $< pairQ2 >$ *results. In Figure 12.1(b),* $/a/d$ *is spilled. Only* $\$a//b$ *and* $\$a/b/c$ *produce results. In both cases, returned pairQ2 elements are partial since they are not composed of all the returned substructures.*

- **Affected path in "where" clause**. When a "where" path falls into SAM, if the missing subpart is not needed for the predicate evaluation, we do not block the predicate evaluation. However, when the "where" path is SRAM, the predicate evaluation cannot be conducted on all the elements. In this case, we may not know whether the results should be output or not. Therefore we treat affected SRAM on the "where" paths as blocking. Whenever a "where" path is SRAM, the output for its corresponding FWR and its inner FWR block thus do not produce anything in our model.

```
Q3: FOR $a in stream()/a
  WHERE $a/d="55"
  RETURN <pairQ3>
     $a/d/f, $a/e, $a/b/c
          </pairQ3>
```

```
Q4: FOR $a in stream()/a
  RETURN <result>$a/c,
      FOR $b in $a/b
     WHERE $b/e ="6"
     RETURN $b/f
        </result>
```

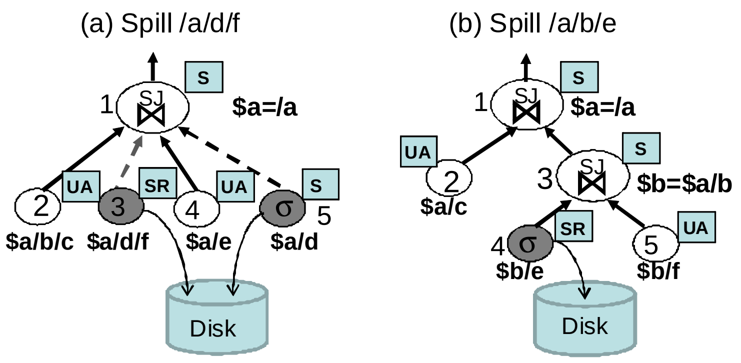


Figure 12.2: Reduced Query Plans for Q3 and Q4

Example 12.2.3. *Query Q3 has a predicate on* $\$a/d$*. Figure 12.2(a) shows the reduced query plan when spilling* $/a/d/f$*. "Where" path* $\$a/d$ *is SAM. In this case, the predicate evaluation is not affected and we can return partial results. Now let us look at Q3 which has a predicate in the inner FWR block. Figure 12.2(b) shows the reduced plan when spilling* $/a/b/e$*. For the inner FWR block, since* $\$b/e$ *is SRAM, the predicate evaluation cannot be conducted. Therefore the inner FWR block cannot produce* $\$b/f$*. However, since* $\$a/c$ *in the outer FWR block is unaffected, we can produce* $\$a/c$ *in the result.*

Chapter 13

Metrics for Quality and Cost

Our optimization goal is to select the optimal paths to spill to maximize output quality. In this work we focus on maximizing the quality of the reduced output. We now describe the metrics of quality and cost for measuring the alternative reduced queries.

13.1 Output Quality Model

Previous studies on approximate query answering tend to focus on the relational model, where the output quality is usually measured by the throughput or the cardinality [10, 59]. However, in our work, since each output result may be partial, measuring the throughput or cardinality of the output is no longer so meaningful. Here we propose a "fine-grained" output quality model which aims to measure the quality of partial XML output results. We measure the quality of the reduced output based on the following factors:

1. **Cardinality**. Since a return structure may be composed of nested substructures, some substructure may only return a subset. So we incorporate the cardinality of each substructure into the output quality.

2. **Shape**. Returned substructures may not be of the full shape when the corresponding

paths in the query fall into SAM. To differentiate such substructures from others, we now define a *shape indicator* to indicate how full each substructure is.

The shape indicator for a path q in the query can be calculated as $S_q = \frac{Size\ of\ element\ after\ spilling}{Size\ of\ element\ without\ spilling}$ (Here we assume the size of an element is fixed).

When a path falls in SAM, its shape indicator is less than 1. In this sense the quality is “punished ” because of returning incomplete substructures.

Recall that the topmost element is the smallest data unit which can produce a result structure. We define *unit quality* as the quality gained by executing the reduced query on a topmost element. We measure unit quality using the formula below:

$$\nu = \sum_{n} \sum_{i=0}^{j} \sum_{q \in B(i)} N_q * S_q \qquad (13.1)$$

Here n indicates the number of return structures generated per topmost element. Each returned structure is composed of j substructures. q denotes the type of nodes matching branch $B(i)$. N_q and S_q denote the cardinality and shape indicator of q, respectively.

Path	**Quality**	
	Spill /a/b	**Spill /a/b/c**
$a//b	1*1	1*1+2*0.5
$a/d	1*1	1*1
$a/b/c	0	0

Figure 13.1: Quality for Q2

Example 13.1.1. *We calculate the unit quality of Q2 for data in Figure 9.3(a) (plan is shown in Figure 1.3). The quality of each substructure is shown in Figure 13.1. For each topmost element a, a result structure <pairQ2> is returned. In this example, only one result structure is produced. Hence n=1. The result structure is composed of three* *substructures,* $\$a//b$*,* $\$a/d$ *and* $\$a/b/c$*. This indicates* j*=3. When spilling path* $/a/b$*,* $d1$ *and* $b3$ *are returned. The unit quality of the reduced query is 1+1=2. When spilling* $/a/b/c$*,* $\$a//b$ *returns three elements,* $b1$*,* $b2$ *and* $b3$*. For* $b1$ *and* $b2$*, their shape indicators are both equal to 0.5 since their* c *children are missing. So the output quality for* $\$a//b$ *is 1+2*0.0.5= 2. The unit quality for Q2 is 1+2=3.*

13.2 Evaluating Reduced Query Costs

We now define a cost model for comparing alternative reduced queries. We measure the cost as the average time of processing a topmost element (we call it the unit processing cost). We divide the processing cost into the following parts: *Locating Cost* (LC) that measures the cost spent on retrieving data and *Join Cost* (JC) spent on structural joins. In addition, in the spilling stage, since we need to flush data to disk, we call the cost spent on spilling data *Spilling Cost* (SC). Since our goal is to optimize the quality of the reduced query, we focus on the cost model of measuring runtime cost savings for the reduced query.

Locating Cost. The locating cost indicates the cost spent on retrieving tokens. Automata are widely used for pattern retrieval over XML streams [22, 44]. The relevant tokens are "recognized" by the automaton and then assembled into elements. The formed elements are passed up to the algebra plan to perform structural join and filtering. While the detailed locating cost model is discussed in [67], we estimate the locating cost savings using the formula below [67]:

$$\sum_{q \in A^{p_i}} n_{active}(q) C_{transit} \tag{13.2}$$

Here P_i indicates the query paths whose subtrees are contained by subtrees of spilled

Notation	Explanation
A^{P_i}	Set of states of pattern P_i and its dependent states.
$n_{active}(q)$	The number of times that stack top contains a state q when a start tag arrives
$C_{transit}$	Cost of transition to states in automaton
N_P	Number of elements matching P for a topmost element
$S_{\bowtie}$	Join Selectivity
M_P	Size of P (number of tokens contained in each element)
C_j	Cost of comparing two elements
$C_{I/O}$	Cost of disk I/O
C_s	Cost of stack operation

Table 13.1: Notations Used in Cost Model

paths. A^{p_i} denotes the set of states corresponding to P_i and its dependent states in the automaton. $n_{active}(q)$ denotes state invoking times and $C_{transit}$ denotes the transition cost. The notations are in Table 13.1.

Join Cost. Since we assume stream data arrives in order, the elements for both join inputs are sorted. We can apply an efficient structural join algorithm, such as Stack-Tree-Anc [3], since both inputs are sorted. Using the cost model for this algorithm [70], we estimate the cost of structural join using the formula as below :

$$2 * N_V N_{B(i)} S_{\bowtie} C_j + 2 N_V C_s \tag{13.3}$$

Here N_V and $N_B(i)$ indicate the number of binding variables and branches per topmost element. Based on Equation 20.3, we can easily calculate the structural join savings for the reduced query.

Spill Cost. Although join computations are saved due to spilling, we now have to consider the additional costs associated with spilling. As will be discussed in Chapter 14, we may have to spill other paths to enable future supplementary result generation. Let us use SP to denote the set of paths to be spilled to disk. The spill cost can then be calculated as follows:

$$\sum_{p \in SP} N_p M_p C_{I/O} \tag{13.4}$$

Runtime Statistics Collection. We collect the statistics needed for the costing using the estimation parameters described above. We piggyback statistics gathering as part of query execution. For instance, we attach counters to automaton states to calculate N_P and $n_{active}(q)$. And we collect M_P and $S_{\bowtie}$ in algebra operators. We then use these statistics to estimate the cost of reduced queries using the formulas given above. Note that some cost parameters in Table 20.1 such as $C_{transit}$, A^{P_i}, C_j and $C_{I/O}$ are constants. We do not need to measure them during the query execution.

Chapter 14

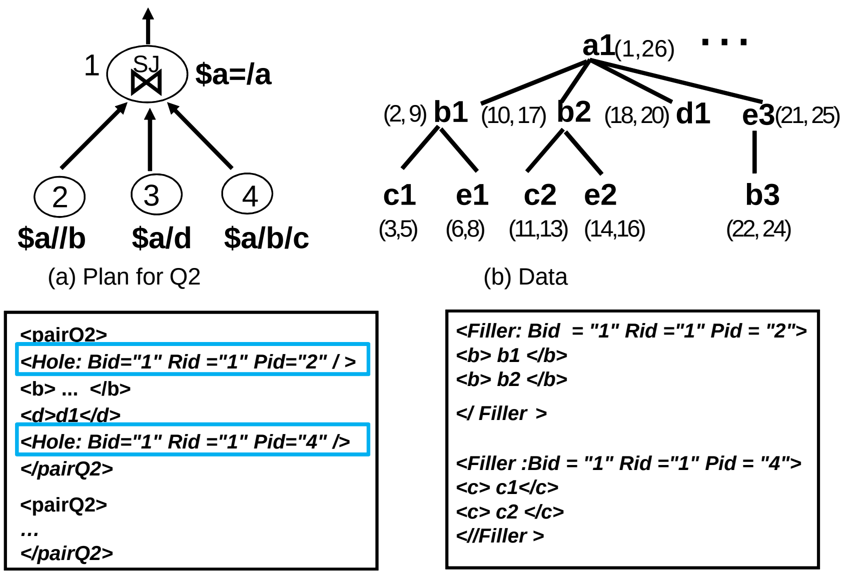
Generate Supplementary Results

In this chapter, we first describe the complementary output model we propose to utilize to match the supplementary "delta" structure with partial reduced outputs produced earlier. Then we examine what extra data must be flushed to guarantee the generation of supplementary results.

14.1 Complementary Output Model

In the clean up stage, supplementary results are generated to "complement" the reduced output produced earlier. So that together these two output "pieces" can be united logically to represent the full content. Since partial result structures may be generated for each output tuple, this requires us to design an output model that can efficiently match the supplementary "delta" structure with the reduced output produced earlier. Here we propose *complementary output model*, which extends from the hole-filler model [25]. The hole-filler model has been designed to organize out-of-order data fragments when an XML document is split into multiple fragments. Our idea is to explicitly mark a hole in the output element with a unique identifier to indicate missing data. In the later cleanup stage, we produce fillers to fill in these holes, which in our context are supplementary results. The reduced outputs and supplementary results for Q2 when spilling $/a/b$ are shown in Figures 14.1(c) and (d) respectively.

To distinguish and match efficiently between holes and fillers, we define three types of IDs, namely, BPU ID (BID), Result Structure ID (RID) and Path ID (PID). Only fillers and holes with the same IDs can be matched. For instance, the first filler in Figure 14.1(d) indicates the missing $b1$ and $b2$ for path $\$a//b$ (whose PID is 2) in the <pairQ2> element for the first BPU (a element). The second filler indicates the missing $c1$ and $c2$ for path $\$a/b/c$ (whose PID is 4) for the first BPU.



```
<pairQ2>
<Hole: Bid="1" Rid ="1" Pid="2" / >
<b> ...  </b>
<d>d1</d>
<Hole: Bid="1" Rid ="1" Pid="4" />
</pairQ2>
<pairQ2>
...
</pairQ2>
```

(c) Reduced Output

```
<Filler: Bid  = "1" Rid ="1" Pid = "2">
<b> b1 </b>
<b> b2 </b>
</ Filler >

<Filler :Bid = "1" Rid ="1" Pid = "4">
<c> c1</c>
<c> c2 </c>
<//Filler >
```

(d) Supplementary Output

Figure 14.1: Example for Output Model

ID	For	Return	ID	For	Return
1	SAM	UA	7	UA	UA
2	SAM	SRAM	8	UA	SRAM
3	SAM	SAM	9	UA	SAM
4	SRAM	UA			
5	SRAM	SRAM			
6	SRAM	SAM			

Table 14.1: Possible Combinations Between For Binding and Its Branches

14.2 Determine Extra Data to Spill for Supplementary Query Execution

To produce eventually complete results set, we have to generate supplementary results correctly. We determine what extra data must be flushed to disk to guarantee the generation of supplementary results. Our goal is to spill a minimum set of data needed for supplementary query execution. The eventual result set must be guaranteed to be both complete and duplicate-free.

Since structural join is the core component in the queries we consider, we focus on how to spill extra data to reconstruct the structural join results correctly. Either the “for” path or the “return” path can be of three types, namely, SRAM, SAM, or UA. There are totally 3*3 =9 combinations between the binding variable and branches. The possible combinations are listed in Table 14.1. Note that if “where” path is SRAM, the output is blocked. Hence we ignore this case.

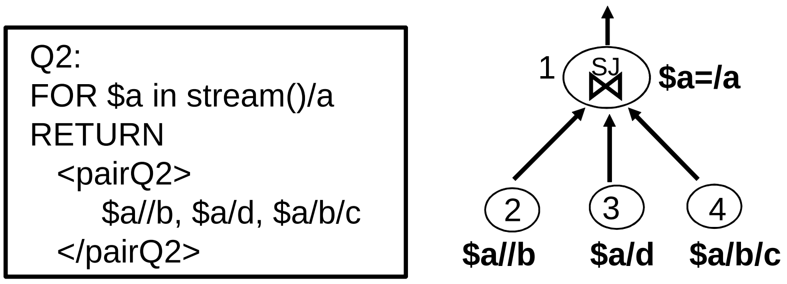
Note when the binding variable is SAM, query execution is not affected. Hence cases 1, 2 and 3 can be regarded to be the same as cases 7, 8 and 9 respectively. Clearly, it is not necessary to consider case 7 since complete results are produced in this case. Finally we only need to consider cases 4-6, 8 and 9. We now describe a typical case, case 8, to show how to determine what extra data to flush to disk and how to compute supplementary results. Similarly, we can generate supplementary results for other cases. The details about those cases can be found in [68].

Binding Variable is UA and Branch is SRAM. In this case, multiple branches may fall into SRAM at the same time. However, the output of the structural join of V with branch $B(i)$ is independent from the output of the structural join between V and other branches. The case that one branch operator falls into SRAM is considered first and can be easily extended to the case that multiple branches are SRAM. Assume that the binding variable V is UA and one branch $B(i)$ is SRAM. We use superscript m and d to distinguish between data kept in memory and data on disk. We represent the structural join results between the binding variable V and $B(i)$ using the following equation:

$$\begin{aligned} V \bowtie_S B(i) &= V \bowtie_S (B^m(i) \cup B^d(i)) \\ &= (V \bowtie_S B^m(i)) \cup (V \bowtie_S B^d(i)) \end{aligned} \tag{14.1}$$

Obviously, the results of $V \bowtie_S B^m(i)$ have already been produced by the reduced query execution. We only need to calculate the supplementary results $V \bowtie_S B^d(i)$. Hence we have to reconstruct the structural join between V and $B^d(i)$ and the extra data to be spilled is the data corresponding to the binding variable V. We use a subscript to indicate the time the data was spilled. Assume that structures V and B have been pushed k times to disk, meaning the spilled data is V_1, V_2, ... V_k and B^d_1, B^d_2, ... B^d_k respectively. As we mentioned in Chapter 10, the query results generated based on a basic processing unit are independent from others. We assume we spill data in batch of one or more basic processing units. We thus conclude that V_x does not need to join with B^d_y if x is not equal to y since they do not belong to the same basic processing unit. Therefore the missing structural join results between V and $B(i)$ at time k can be calculated as $V_k \bowtie_S B^d_k(i)$.

For instance, for the plan of Q2 in Figure 14.2, when path $/a/b$ is spilled, path $\$a//b$ is SRAM. The structural join between $\$a$ and $\$a//b$ can be calculated using Equation 14.1.



```
Q2:
FOR $a in stream()/a
RETURN
   <pairQ2>
        $a//b, $a/d, $a/b/c
   </pairQ2>
```

(a) **Query Q2** (b) **Query Plan**

Figure 14.2: Query Q2 and Its Plan

Chapter 15

Choose the Optimal Structure to Spill

15.1 Formulation of Optimization Problem

For each spill candidate, a reduced query is derived to produce the reduced output. For each reduced query, we measure its unit quality and unit processing cost. Unit quality for a reduced query is defined as the quality gained by executing the reduced query on a topmost element. Unit processing cost is the average time of processing a topmost element. Our goal is to pick structures to spill so as to optimize the output quality. The problem can be formulated as follows. Given the following inputs: 1. data arrival rate λ in the number of topmost elements per time unit, 2. unit quality gained by executing each reduced query $\{\nu_0, \nu_1, ..\nu_n\}$, 3. unit processing costs for each candidate reduced query $\{C_0, C_1, ..C_n\}$. We aim to find a spill candidate whose corresponding reduced query satisfies the following two conditions: (1) consume all input elements in 1 time unit, and (2) maximize total output quality.

Given a spill candidate, we first derive its corresponding reduced query Q_i. We use $1/C_i$ to calculate how many elements can be processed when executing Q_i per time unit. Since the processed data cannot exceed the incoming data, the total output quality is calculated using the formula below:

$$\nu_i * min\{\lambda, 1/C_i\} \tag{15.1}$$

15.2 Algorithms for Spill Optimization

Optimal Reduction(OptR). The first algorithm we propose, called Optimal Reduction (OptR), employs an exhaustive approach. It searches the entire candidate space and picks the candidate that yields the highest output quality.

The procedure proceeds as follows: 1) iterate over each spill candidate in a top-down manner in the candidate lattice and derive a reduced query Q_i. and 2) then estimate the cost, unit quality as well as total output quality of Q_i. The candidate query that has the highest output quality will be chosen as the reduced query at the spilling phase.

Remember from Chapter 11 that f is the fan-out and d is the depth of the query pattern tree. Since it is an exhaustive approach, the search complexity is equal to the size of the minimum candidate space, which is $O(2^{fd})$.

Example 15.2.1. *Assume the arrival rate is 20 topmost elements/s. The unit cost and unit quality for the initial query are 0.1s and 6 respectively. The available CPU resources are 1 second. In this case, the reduced query needs to process 20 topmost elements while achieving the highest output quality. The unit processing cost and unit quality for each candidate are shown in Figure 15.1. We pick spill candidate $\{b, c\}$ since its corresponding reduced query yields the highest output quality, namely, (1 / 0.05) * 2 = 40.*

Optimal Reduction with Pruning (OptPrune). Optimal Reduction with Pruning (OptPrune) applies additional pruning to eliminate suboptimal solutions. It explores the spill candidate space in a top-down manner and removes less promising solutions based on the observation below.

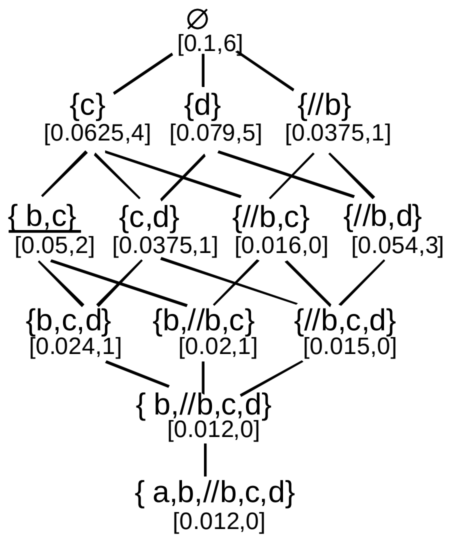


Figure 15.1: Optimization Using OptR

Observation 15.2.1. *In the top-down candidate space traversal, when we reach a candidate d_i and find it is capable of consuming all input data, then the candidates below it (candidates that include all paths in d_i) can all be pruned.*

The reason is that if candidate d_i can produce r_i result structures, the candidates below it tend to spill more paths. The quality of each result structure is not higher than that of candidate d_i. However, the number of output result structures may stay unchanged since all input data is consumed. Therefore, the total quality of the candidate below d_i is guaranteed not to be higher than that of d_i.

Example 15.2.2. *In Figure 15.2(a), candidate $\{b, c\}$ can consume all input. In this case, we can prune candidates below it, $\{b, c, d\}$, $\{b, //b, c\}$ and $\{b, //b, c, d\}$ directly. Similarly, candidates below $\{//b\}$ and $\{c, d\}$ can be removed.*

To estimate the search complexity, since the worse case for OptPrune is checking every candidate without pruning anything, therefore the worst case for OptPrune is $O(2^{fd})$. However, our experimental results will show that the actually complexity is much smaller

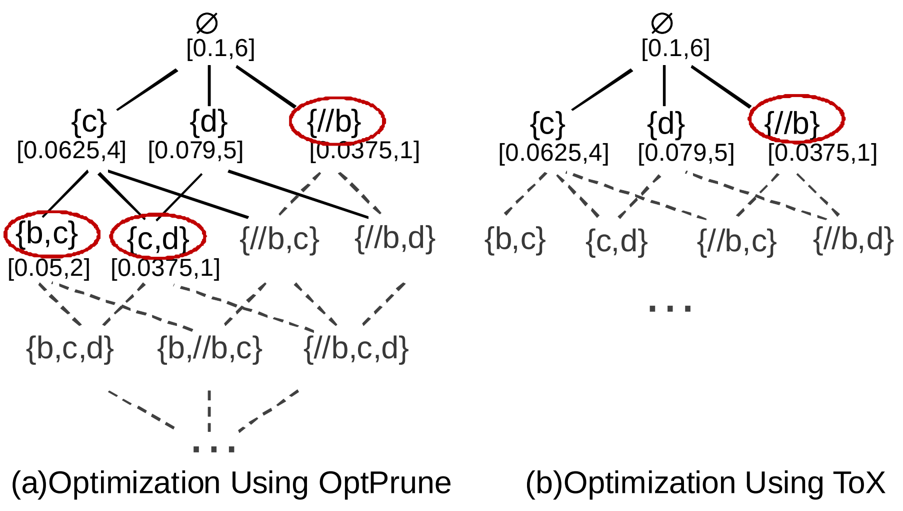


Figure 15.2: OptPrune and ToX Example

than $O(2^{fd})$.

Top-down Expansion Heuristic (ToX). We now present a Top-down eXpansion heuristic (ToX), which has much more efficient running time compared to OptR and OptPrune. ToX starts from simple spill candidates and stops at the first candidate that is able to consume all the input.

ToX proceeds as follows:

Step 1. Check candidates that spill one leaf node (candidates on the top level of the lattice). If we find a candidate that is able to consume all input and achieve highest total output quality among candidates considered so far, stop. Otherwise go to Step 2.

Step 2. Pick the candidate that has the highest quality/cost ratio on this level and move to candidates connecting it one level lower.

Step 3. If one of the new candidates can consume all the input and achieve the highest total output quality among candidates considered so far, stop. Otherwise go back to Step 2.

The complexity of ToX is $O(f^{2d})$ which is much smaller than that of OptR and OptPrune.

Example 15.2.3. *In Figure 15.2(b), we first check the candidates that only spill one node. We find* $\{//b\}$ *can consume all input. We consider* $\{//b\}$ *optimal and stop. The total output quality is* $min\{$*20, 1/0.0375*$\}$ ** 1 = 20.*

Chapter 16

Experimental Evaluation

In this chapter, we conducted a comparative study of the three optimization algorithms OptR, OptPrune and ToX. In addition, we also employed an algorithm, called *Random*, which iteratively selects one among all possibly substructures randomly until enough substructures are spilled so that the input load can be handled by the corresponding reduced query. The experimental results demonstrated that our proposed solutions consistently achieved higher quality compared to the Random approach. The experiments are divided into three categories:

- The first set of experiments compared the performance of our proposed spilling strategies with Random approach in two cases. One case is when the network is fast and reliable, i.e, the input sources are never blocked. The other case is when the network is unreliable. When the network is unreliable, the input data has a mixture of high and low arrivals. When the arrival rate is low, the disk on data can be processed and generate supplementary output.

- The second set of experiments tested the impact of different selectivity and different query path sizes on the performance of our approaches.

- The third set of experiments compared the overhead of different spilling approaches.

Experimental Setup. We have implemented our proposed approaches in an XML stream system called Raindrop [30]. The data sets were generated using ToXgene [12]. All experiments were run on a 2.8GHz Pentium processor with 512MB memory.

16.1 Comparison of Spilling Approaches

16.1.1 Reliable Networks

A reliable network never incurred suspensions of data transmission. For achieving this, we set arrival interval between two topmost elements to a fixed value. In this set of experiments, we set arrival interval to 0.025s and 0.02s respectively. The arrival rates under these two settings were higher than the processing speed. We used Q2 as the running query. Spilling was invoked as soon as the memory buffer threshold is reached.

To compare the performance of alternative approaches, we used a new “fine-grained” quality metric to measure the quality of partial outputs instead of using traditional throughput metric. The reason is that throughput typically refers to the number of (complete) output elements in XML produced. However, in this work of producing partial structures, a traditional throughput metric is not so meaningful. The detailed quality model can be found in Chapter 13.

We studied the output quality gained by taking different optimization approaches. Figures 16.1 and 16.2 show the cumulative output quality using four optimization strategies when the arrival interval is 0.025s and 0.02s respectively. Observe that OptR, OptPrune and ToX gained higher quality than Random after spilling starts. OptR and OptPrune both gained higher quality than Random and ToX. This is because OptR and OptPrune guarantee to find the optimal structures to spill.

Because the reliable network never incurred suspension of data transmission, the clean up processing was invoked after all the data has arrived (after time 5500). In the clean up phase, the supplementary results were generated based on the disk-resident data. Finally all four spilling approaches produced the complete result set and reached the same output quality.

When the arrival interval is 0.02s, the cumulative quality increased slower than the case that the interval is 0.025s. This is because when the arrival rate was increased, the reduced query may need to spill more structures to consume all the input.

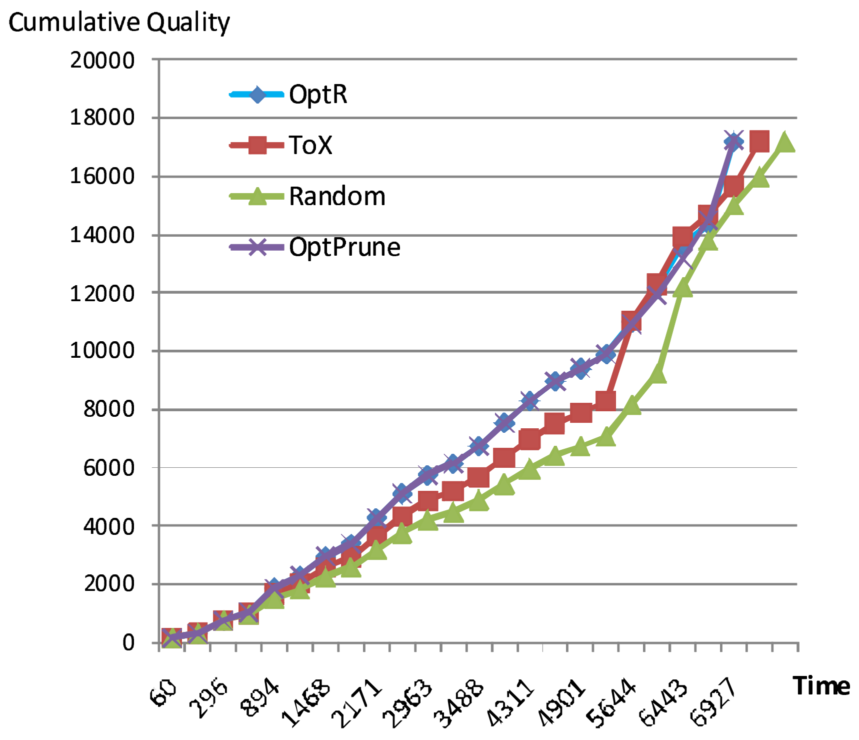


Figure 16.1: Reliable Network, 0.025s

16.1.2 Unreliable networks

Having evaluated our spilling approaches in the absence of transmissions, we proceed to examine the performance for unreliable networks. To simulate unreliable network, we

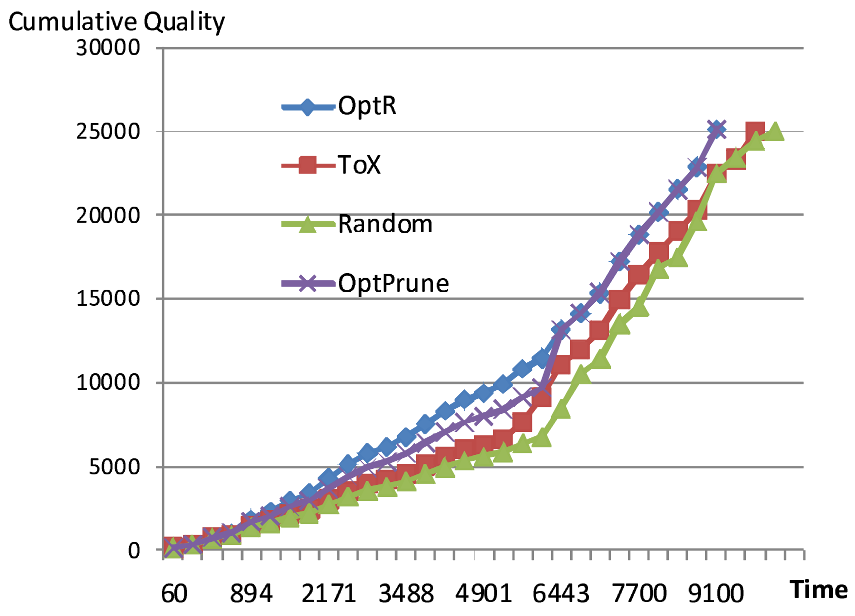


Figure 16.2: Reliable Network, 0.02s

generated arrival intervals using Pareto distribution that is widely used in case of bursty network [20]. Figure 16.3 shows the cumulative quality for four approaches. Observe that all of them had step-like performance due to switching between the spilling and clean up phase. The slope of segments corresponding to the spilling phase for OptR and OptPrune was larger than that of ToX and Random. This indicates that output quality for OptR and OptPrune is increased faster than that of ToX and Random.

16.2 Impact of Selectivity and Path Size

Next, we illustrated that the output quality is affected by the selectivity distribution of the binding variable and each branch. We ran the query Q5 below:

```
Q5: FOR $o in stream("test")/list/o
 RETURN $o/P1, $o/P2, $o/P3, $o/P4
```

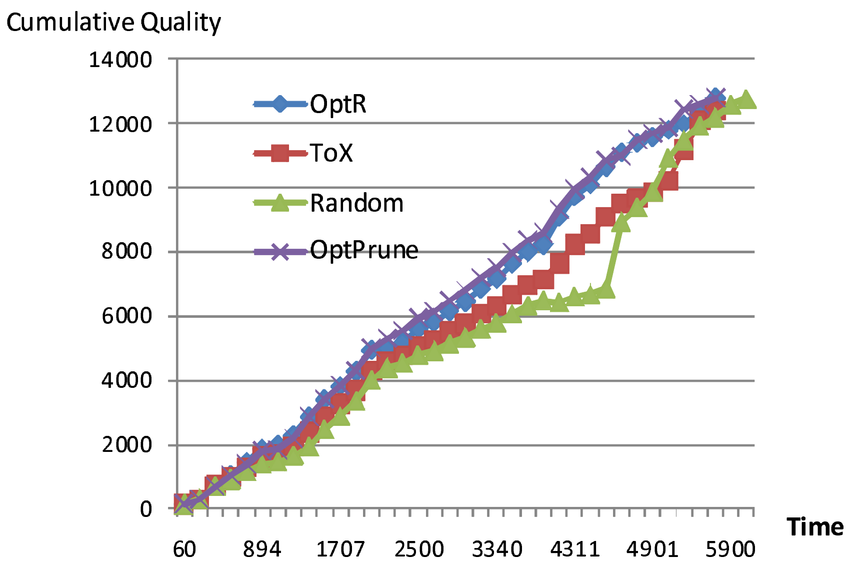


Figure 16.3: Unreliable Network

We generated five test data sets which satisfy the following requirements: 1) all test data sets contained the same number of tokens; and 2) the numbers of elements corresponding to each returned path were equal; and 3) the element sizes corresponding to each returned path were equal. Based on the cost model in Chapter 13.2, the locating costs spent on locating each returned path are the same. The join costs between the binding variable and each returned path are the same too. In addition, the spilling costs when spilling each returned path are also the same. For each data set, the selectivity between the binding variable and its branches can be different. We used five different sets of selectivity which differ in their standard deviations. Figure 16.4 shows that the output quality is higher when there is a bigger variance among selectivity for OptR and OptPrune. This is because OptR and OptPrune tend to spill the return paths with low selectivity which yield low output quality given the same spilling and computation cost. We observe that the quality of the reduced query achieved by the Random approach did not change a lot because Random approach did not keep the returned paths having large selectivity.

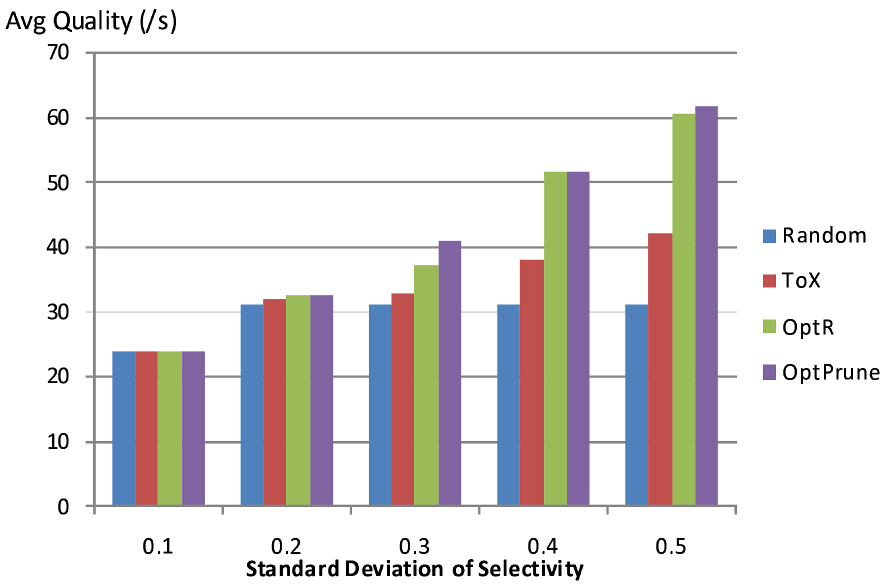


Figure 16.4: Quality for Varying Selectivity

We now illustrate that the output quality is affected by the pattern size. All testing data sets had the same number of elements and selectivity for each returned paths. And all test data sets contained the same number of tokens. Figure 16.5 shows that the output quality changes with varying standard deviation of return path size. For the Random approach, the output quality did not change a lot. However, for OptR and OptPrune, the output quality was much higher than the quality achieved by Random approach when the standard deviation of pattern size increased. This is because the reduced queries with smaller returned path size have smaller spilling cost, resulting in lower overall processing cost. In this case, OptR and OptPrune would pick such reduced queries since they have relatively higher quality/cost ratios and thus higher quality.

16.3 Overhead of Spilling Approaches

In this work, optimization is conducted in an online fashion to assure continuous responsiveness of our system. Here we studied the overhead of four spilling strategies, measured

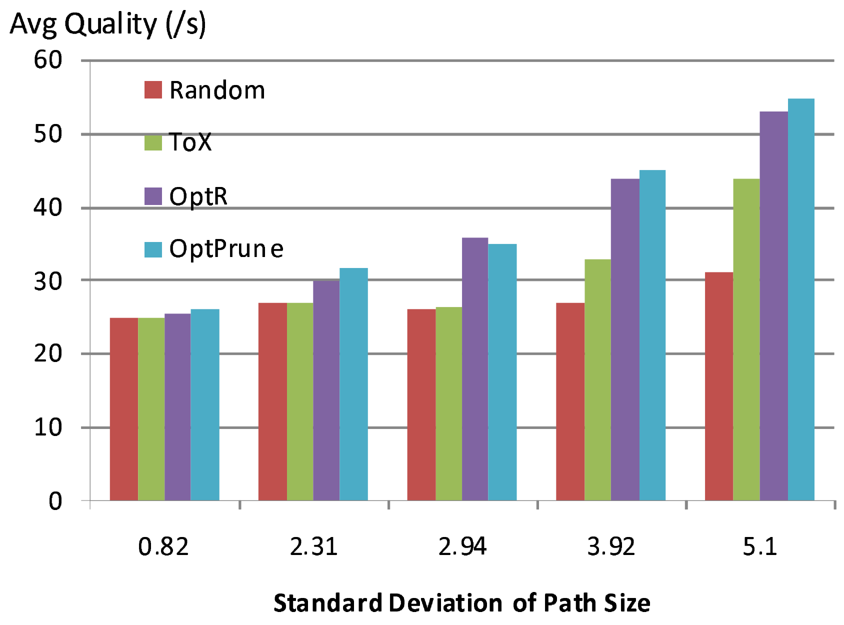


Figure 16.5: Quality for Varying Path Size

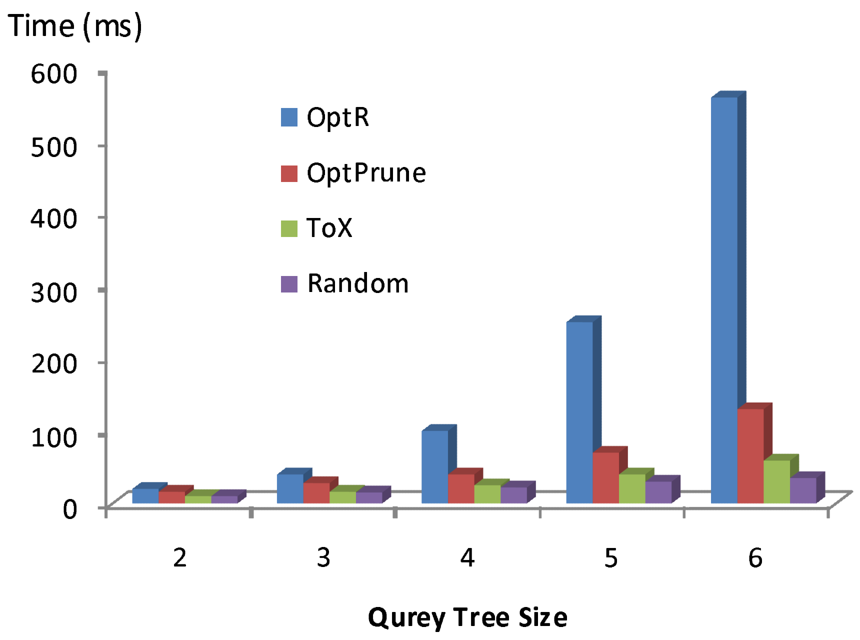


Figure 16.6: Overhead of Four Approaches

by the time spent on choosing which structure to spill. We studied the relationship between the complexity of the query and the overhead of the optimization methods. We used five queries which vary in the size of the query trees. In Figure 16.6, when the queries became complex, the overhead of ToX was much smaller than OptPrune and OptR since it stopped at the earliest candidate which consumes all input. We observe that the overhead of OptPrune was much smaller than that of OptR. This indicates that our pruning method is indeed effective at reducing the search cost. Given that both approaches can achieve the highest quality, OptPrune is obviously a better option than OptR. However, when the query became more complex, OptPrune may not be a practical solution since its overhead is larger than ToX and Random. In this case, we resolved to utilize our lightweight ToX solution.

Chapter 17

Related Work

Complete result sets are often required for continuous queries, even though the query system may not have sufficient resources for the query workload at a particular time. To address this, prior works have focused on flushing data temporarily to disk to address the problem of run-time main memory shortage while satisfying the needs of complete query results for relational streams. Most of them are focused on maximizing the output rate or generating a subset of results as early as possible [39, 48, 62, 63].

[63] is the first to propose a non-blocking join operator, called XJoin, which produces results event when one or more stream sources experience delays. [63] proposes to conduct hash join during three stages. The first stage joins memory resident tuples, acting similarly to the standard symmetric hash join. The second stage joins tuples that have been flushed to disk due to memory constraints. The third stage is a clean-up stage that makes sure that all the tuples that should be in the result set are ultimately produced. Hash-Merge Join [48] proposes a Hash-Merge join approach which produces non-blocking output by employing an in-memory hash-based join algorithm at run time and employing a sort-merge-like join algorithm in the merging phase. [23] proposes a non-blocking sort-merge join approach to produce joined output which eliminates the blocking behavior of sort-

based join algorithms.

[41] designs a PermJoin approach for producing early results in multi-join query plans. [41] aims to maximize the early overall throughput and to adapt to fluctuations in data arrival rates. [41] employs a flushing policy to write in-memory data to disk, once memory allotment is exhausted, in a way that helps increase the probability of producing early result throughput in multi-join queries. [43] tackles the query plan with multiple state intensive operators where data spilling from one operator can affect other operators in the same pipeline. We can apply the above techniques on coarse-grained spilling in XML, which is spilling complete topmost elements to disk. However, such coarse-grained spilling misses the novel XML-specific opportunities for spilling. In this work, we instead focus on the fine-grained XML-specific structural spilling approach.

Niagara [54] proposes to produce approximate results for XQuery when no input for some operators in the plan exists. However, they do not address the problem of producing partial results in the XML stream context when the output from one operator is missing due to spilling some patterns. Part I of this dissertation addresses structural shedding problem in XML streams. However, it only considers queries containing independent returned paths. Also, since it is focusing on shedding, how to generate supplementary results is not discussed.

[15, 22, 29, 33, 44, 52] evaluate XQuery expressions over XML streaming data. One approach [22, 33] combines automaton and algebra to process XML stream data. E.g., Tukwila [33] and YFilter [22] model the whole automaton processing as one mega operator while modeling the rest data manipulation such as filtering and restructuring in algebraic operators. [15, 29, 44, 52] use automata or automaton-like SAX event handlers to process the whole query. One limitation of our structural spilling framework is that the cost model measuring processing costs is related to the specifics of the implementation of query processing. Therefore, we can apply our spilling techniques to other XML stream systems as long as we plug in their cost models.

Part III

An Integrated Framework for Structural Shed and Spill Approach

Chapter 18

Overview of Our Approach

The architecture of our integrated framework is shown in Figure 18.1. After the queries are registered with the query engine, an initial plan is generated and optimized. The execution engine will instantiate the query plan and start processing input streams. The problem of deciding when the system needs to shed or spill data is not a question specific to XML stream. Any existing approach from the literature [48, 63] could be employed here. We employ a memory buffer to store input stream data. As soon as a token is processed, we clean this token from the buffer. We assume a threshold on the memory buffer that allows us to endure periodic spikes of the input. When buffer occupancy exceeds the given threshold, we trigger the optimization process.

Each fusion candidate corresponds to a pair of a reduced query and its matching supplementary query. The *reduced query*, which is devised from the initial query by reducing some computations, is executed when the arrival rate is high. When the arrival rates become slow, *supplementary query* is executed to produce the output that complements the partial output produced by the reduced query earlier. When the load reduction process is triggered, the possible fusion candidates are examined. The query optimizer runs the optimization algorithm to pick the optimal reduced and supplementary query pair. After

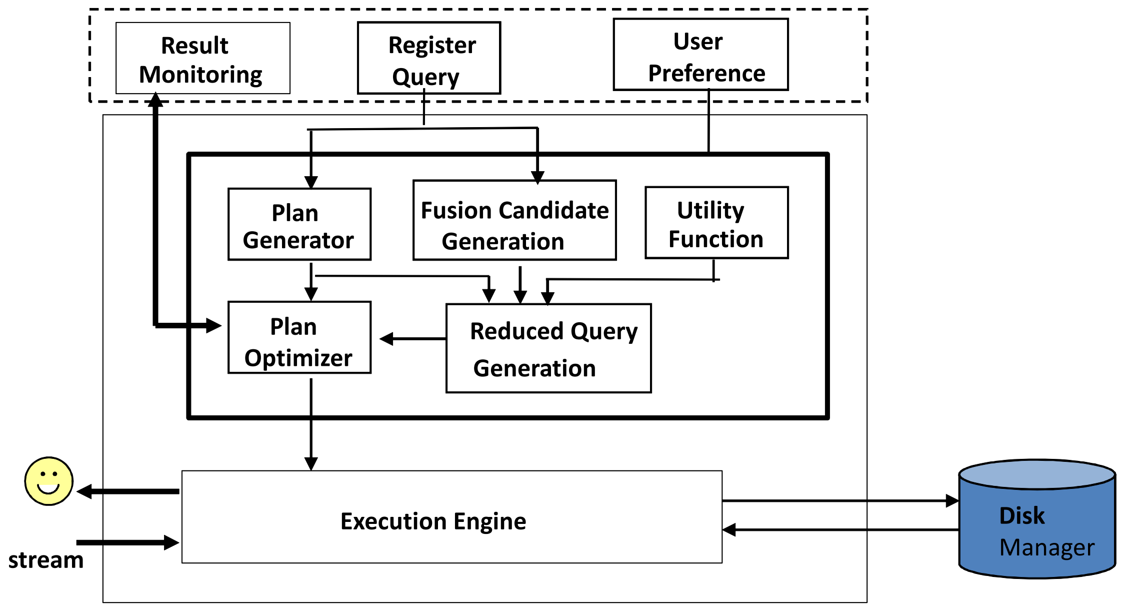


Figure 18.1: Architecture for Integrated Framework

optimization, the reduced query is instantiated. The reduced query takes the place of the previously active query, initiating the shedding or spilling processes whenever needed. When arrival rates become slow and the system has enough resources to execute both the original user query and the supplementary query, we retrieve the data back from disk and then the supplementary query is executed.

In this work, we assume that any path and any number of paths in the query can be shed or spilled. We describe the space of possible fusion candidates in Chapter 19. Before illustrating how to pick the optimal fusion candidate, we define *feasible fusion candidate* and *feasible query* as follows.

Definition 1. *For a fusion candidate, if its reduced query can consume all the inputs, i.e., the processing speed of the reduced query is faster than or equal to the arrival rate, we call this fusion candidate feasible and the reduced query a feasible query.*

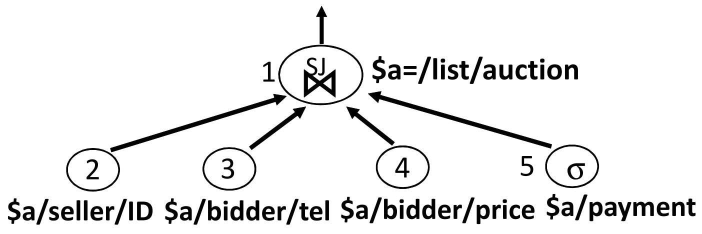
Our optimization goal is to pick the optimal fusion candidate that produces the highest total output quality, and assure that the fusion candidate is feasible.

Let us use the fraud detection query introduced in Chapter 1.4.3 as our example (the query and its plan are shown in Figure 18.2). Assume we have two fusion candidates for fraud detection query Q1. Fusion candidate 1 $\{\$a/bidder/price^D$, $\$a/bidder/tel^P\}$ sheds $price$ permanently and spills tel to disk. The superscripts indicating the actions on substructures. D indicates shed and P indicates spill. The reduced and supplementary queries for candidate 1 are shown in Figure 18.4. Fusion candidate 2 $\{\$a/bidder/tel^D$, $\$a/bidder/price^P\}$ sheds tel and spills $price$ to disk. The input stream data fragment for Q1 is shown in Figure 18.3. Figure 18.5 shows the output produced by the reduced and supplementary queries for these two fusion candidates. The arrival pattern for Q1 is shown in Figure 18.6. The arrival rate is 500 auction elements/second from time 0 to 1s and zero from 1s to 2s. Our goal is to pick an optimal feasible fusion candidate to maximize the total output quality. Assume substructures $\$a/bidder/tel$ and $\$a/bidder/price$ are of the same size. In this case, the shedding costs for $\$a/bidder/tel$ and $\$a/bidder/price$ are the same. The spilling costs for these two query paths are the same too. Assume the costs to produce each partial <pairQ1> element at runtime for fusion candidates 1 and 2 are both 1.6 milliseconds (the detailed cost measurement can be found in Chapter 20.2). The processing rates for both fusion candidates are 1000/1.6 =625. Both values are greater than the arrival rate. Therefore both fusion candidates meet our requirement of feasible candidates. In addition, the unit quality for early partial <pairQ1> is both 1 (the quality measurement can be found in Chapter 20.1). Therefore, both two candidates achieve the same reduced output quality.

Now let us look at the supplementary query. The costs of producing supplementary output for two fusion candidates are the same because elements tel and $price$ are of the same size. Let us assume that the costs of producing supplementary output $\$a/\ bidder/\ tel$ and $\$a/bidder/price$ are both 0.5 milliseconds. However, the quality gained during lull time periods for these two fusion candidates differs. Based on Figure 18.6, the quality

```
Q1: FOR $a in stream()/list/auction
      WHERE ($a/seller/payment [contains(., "escrow service")])
          and ($a/seller/payment[ not(contains(., "Escrow.com")])
      RETURN <pairQ1>
                    $a/seller/ID, $a/bidder/tel, $a/bidder/price
             </pairQ1>
```

(a) **Query Q1**



(b) **Query Plan for Q1**

Figure 18.2: Query Q1 and its Plan

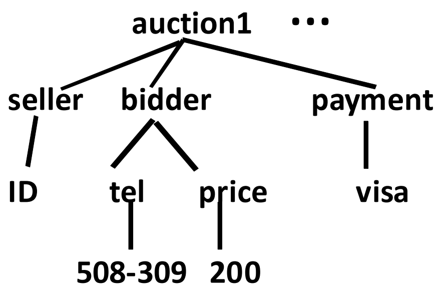
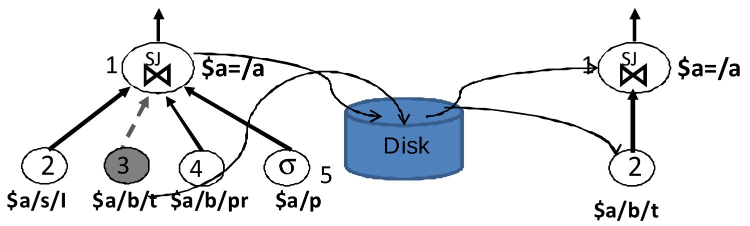


Figure 18.3: Data Fragment for Q1



(a) Reduced Plan (b) Supplementary Plan

Figure 18.4: Reduced and Supplementary Queries for Candidate 1

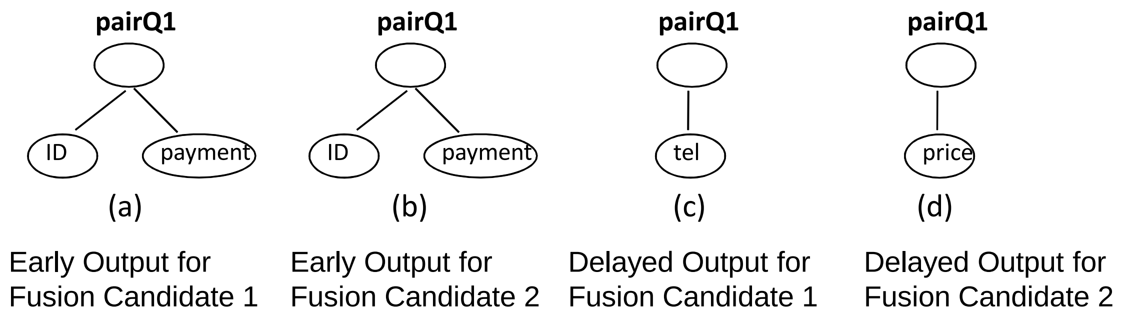

pairQ1
ID
payment
(a)
Early Output for Fusion Candidate 1
pairQ1
ID
payment
(b)
Early Output for Fusion Candidate 2
pairQ1
tel
(c)
Delayed Output for Fusion Candidate 1
pairQ1
price
(d)
Delayed Output for Fusion Candidate 2

Figure 18.5: Early and Delayed Output for Q1

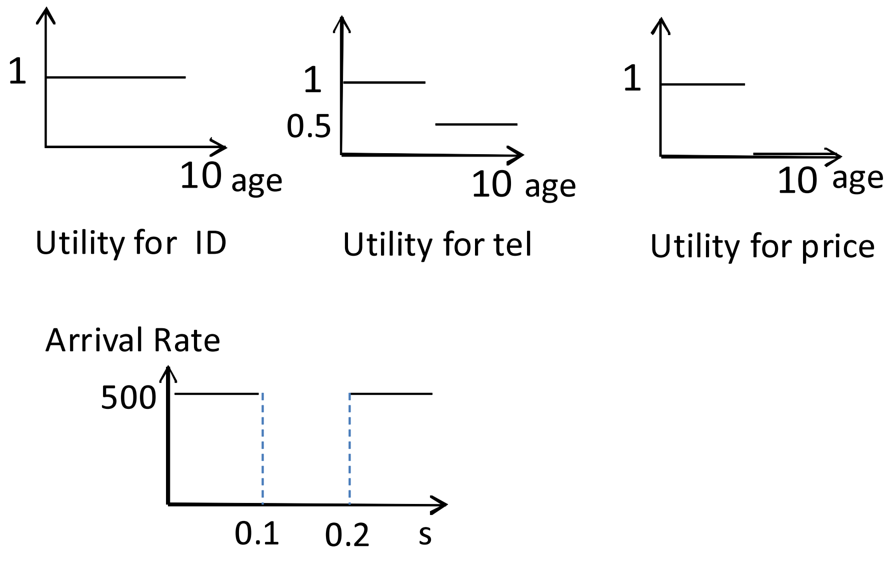


Figure 18.6: Utility Function For Each Path and Arrival Pattern

for $\$a/\ bidder/\ tel$ is 0.5 since its quality sheds 50% if its delivery is delayed. However, the utility for delayed output $\$a/\ bidder/\ price$ drops to zero. Therefore during the lull time period the delayed output $\$a/\ bidder/\ tel$ has higher quality than $\$a/bidder/price$. In this case, we choose candidate 1 $\{\$a/bidder\ /price^D$, $\$a/bidder/tel^P\}$ since its total output quality is higher than that of candidate 2. We will describe the algorithms to find an optimal fusion candidate in Chapter 21.

Chapter 19

Fusion Candidates

In this section, we first give possible delivery options. Then we describe the representation of possible fusion candidates.

19.1 Delivery Options

When data arrival speed is extremely high and the system resources are limited, partial results instead of complete results may have to be produced. Since XML results are composed of various substructures, output substructures may vary in their delivery time. For each substructure, we have following three options to handle data:

- **Keep**. The first option is to deliver the substructure by processing and reporting immediately. We label this substructure “Keep.”
- **Spill**. The second option is to push data to disk temporarily. The data on disk will be brought back when sufficient resources are available in the future. We label this substructure “Spill.”
- **Shed**. The third option is to permanently throw the data away when the arrival rate is high. We label such substructures “Shed.”

19.2 Representation of Fusion Candidates

In this section we examine all possible fusion candidates. Here a query is represented by a query pattern tree. For example, the query pattern tree for Q1 is in Figure 19.1. In query pattern tree, each navigation step in an XPath is mapped to a tree node. We use single line edges to denote parent-child relationships.

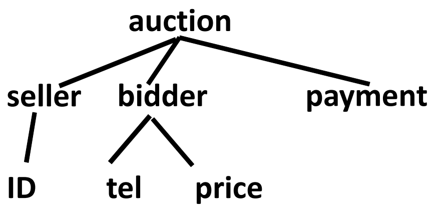


Figure 19.1: Pattern Tree for Q1

We assume any node in the query tree can be shed, spilled, or kept. Since each node in the query pattern tree has three options, namely, keep, shed or spill, the combination of selecting one of these options for each node in the query tree represents a fusion candidate. For instance, $\{auction^K$, $seller^K$, $bidder^K$, $payment^K$, ID^K, tel^P, $price^P$ $\}$ is a fusion candidate. We use a vector, whose length is equal to the total number of nodes in the query tree, to represent a fusion candidate. For each node in the query tree, there is a corresponding value in the vector indicating the action conducted on this node. 0, 1, and 2 indicate "keep," "spill" and "shed" respectively. The position for each node in the vector is fixed and follows the node's preorder in a tree traversal. For instance, for query Q1, the preorder traversal follows the order $auction \rightarrow seller \rightarrow bidder \rightarrow payment \rightarrow ID \rightarrow tel \rightarrow price$. Vector $[0, 0, 0, 0, 0, 1, 1]$ represents fusion candidate $\{auction^K$, $seller^K$, $bidder^K$, $payment^K$, ID^K, tel^P, $price^P\}$. For readability, we keep a letter in the vector to remind readers about the label of each node. So we use $[a^0, s^0, b^0, p^0, i^0, t^1, r^1]$ to represent $[0, 0, 0, 0, 0, 1, 1]$. Thus the action is indicated on superscripts.

19.3 Solution Space of Fusion Candidates

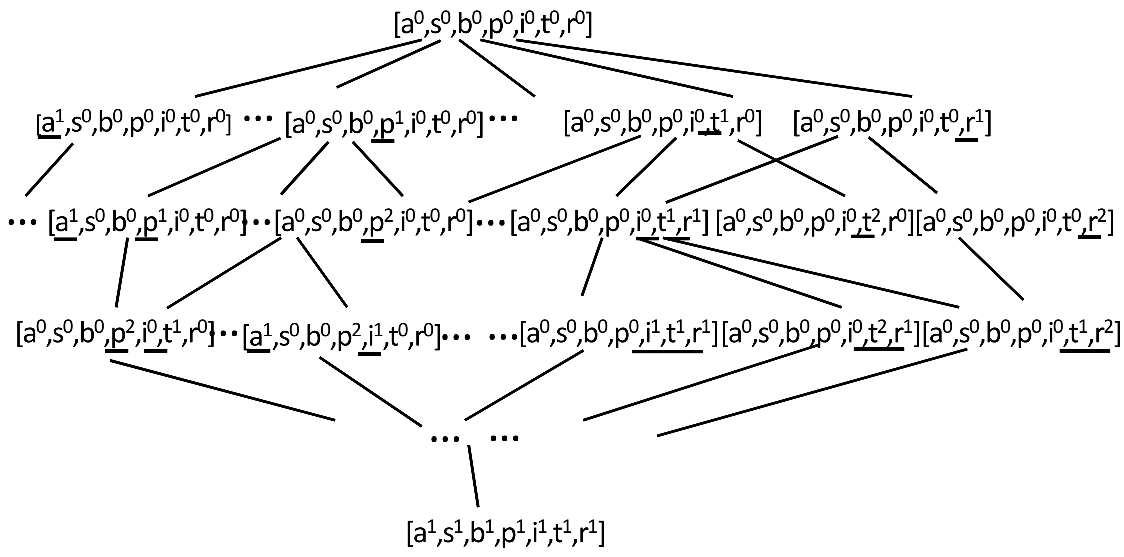

[a0,s0,b0,p0,i0,t0,r0]
[a1,s0,b0,p0,i0,t0,r0] ··· [a0,s0,b0,p1,i0,t0,r0] ··· [a0,s0,b0,p0,i0,t1,r0] [a0,s0,b0,p0,i0,t0,r1]
··· [a1,s0,b0,p1,i0,t0,r0] ·· [a0,s0,b0,p2,i0,t0,r0] ··· [a0,s0,b0,p0,i0,t1,r1] [a0,s0,b0,p0,i0,t2,r0] [a0,s0,b0,p0,i0,t0,r2]
[a0,s0,b0,p2,i0,t1,r0] ·· [a1,s0,b0,p2,i1,t0,r0] ··· ···[a0,s0,b0,p0,i1,t1,r1] [a0,s0,b0,p0,i0,t2,r1] [a0,s0,b0,p0,i0,t1,r2]
··· ···
[a1,s1,b1,p1,i1,t1,r1]

Figure 19.2: FC Lattice for Q1

We now design a lattice to represent all possible fusion candidates, i.e., the FC search space. Each fusion candidate corresponds to a reduced and supplementary query pair. For instance, $[a^0, s^0, b^0, p^0, i^0, t^0, r^0]$ indicates that every node is kept during the reduced query processing. In other words, the reduced query is the original query and the supplementary query is an empty query. Fusion candidate $[a^2, s^2, b^2, p^2, i^2, t^2, r^2]$ located at the bottom of the lattice indicates every node is shed during the reduced query processing. In this case, not only is the reduced query an empty query, but the supplementary query is also an empty query since everything is dropped. From level i to level i+1, only one node changes its associated action and the action change of a query tree node follows the order from “0” (“Keep”) to “1” (“Spill”) or from “1” (“Spill”) to “2” (“Shed”).

To analyze the total number of potential fusion candidates, consider a complete query pattern tree q with depth d and fixed fan-out f. The total number of nodes in this query tree can be calculated as: $\tau = \sum_{i=1}^{d-1} f^i = \frac{f^d - 1}{f-1}$. Since each position in vector can have three values, “0,” “1” or “2”, and the vector length is equal to the total number of nodes in the query tree, the total number of fusion candidates can be calculated as 3^{τ}, which is bounded by $O(3^{f^d})$. The number of levels in the FC lattice is 2τ since each level increments in 1 position and the initial vector $[a^0, s^0, b^0, p^0, i^0, t^0, r^0]$ needs to increment 2τ times before every value increments to 2. For a fusion candidate, the number of its direct child candidates in the lattice is less than or equal to τ since the action for every node in the vector may change until it reaches 2.

Based on our cost model in Chapter 20.2, we observe that when we change the action of a node from “Keep” to “Spill” or from “Spill” to “Shed”, the cost for producing the reduced output is decreased. This leads to the following observation.

Observation 19.3.1. *For a fusion candidate FC_i on level i, the descendant fusion candidates on level $i+1$ or below are guaranteed to have smaller reduced query processing costs compared with candidate FC_i.*

Chapter 20

Metrics for Quality and Cost

20.1 Quality Model

Our objective is to maximize the output quality of the reduced output as well as the supplementary output while staying within given system resources. We now describe the metrics of quality for measuring alternative reduced queries as well as their respective supplementary queries.

20.1.1 Age-based Utility Specification

For stream applications, the quality may be affected when the delivery of output results is delayed. If the output is delayed for a smaller period of time, it is likely more acceptable to users compared to longer or even indefinite delays. In fact, if the delay is too long, the data may become useless to users. Hence we need a metric to evaluate to what degree output quality is affected. Relational stream systems [8] proposed a latency-based QoS graph where a piece-wise linear function is specified to indicate the latency-based utility for a query. For XML stream data, since output may be composed of multiple substructures, we instead propose to use a corresponding utility function to indicate the utility of each

XML substructure. Here, output elements are treated as atomic units. Similar to [8], the utility function is a piece-wise linear function with the following properties: 1) maximum utility at time zero, 2) incomplete utility value when the age gets older, and 3) a deadline latency point after which output data provides zero utility. A utility function is defined as follows:

$$\mu = \begin{cases} \mu_0 & \textbf{if} \quad t < t_0 \\ \cdots & \\ \mu_i & \textbf{if} \quad t_{i-1} \leq t < t_i \\ \cdots & \\ \mu_n & \textbf{if} \quad t_{n-1} \leq t < t_n \end{cases}$$

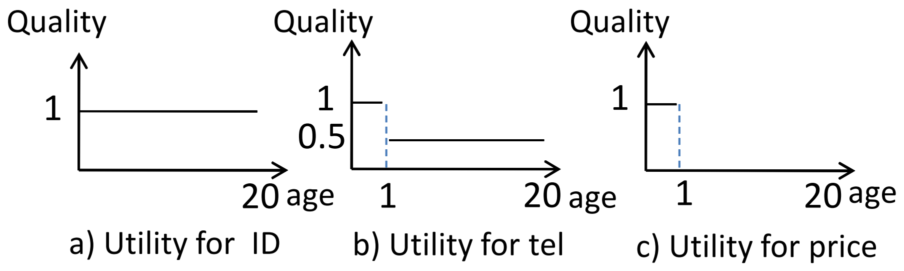


Figure 20.1: Utility Function Examples

An example age-based utility function is shown in Figure 20.1(b) and (c). Note that the utility for both $\$a/bidder/tel$ and $\$a/bidder/price$ has the maximum utilty 1 at time zero. The utility of $\$a/bidder/tel$ drops to 0.5 if its delivery is delayed by 5 seconds, while the utility for delayed output $\$a/bidder/price$ drops to zero if its delivery is delayed by 5 seconds. Without loss of generality, we assume the utility for a query path is normalized to [0,1].

20.1.2 Output Quality Computation

Previous studies on approximate query answering [10,59] tended to focus on the relational model, where the output quality is measured by either the output rate or the cardinality.

However, since each output result may be partial, measuring the cardinality of the output as a simple count without considering its output complexity is not sufficient for the data. Here we propose a "fine-granularity" output quality model that aims to measure the quality of partial XML output results. Our measure is based on the following factors:

1. **Cardinality**. For XML data, a return structure may be composed of multiple substructures. We incorporate the cardinality of each substructure into the output quality.

2. **Age-based utility**. We consider age-based utility for each substructure as one key factor of the quality computation of partial XML results.

3. **Shape**. Returned substructures may not be of their complete shape when some part of the substructure is missing. For example, suppose both paths $\$a/b$ and $\$a/b/c$ are in the "return " clause. When path $\$a/b/c$ is spilled to disk, the path $\$a/b$ would return incomplete b elements. To differentiate such substructures from others, we define a *shape indicator* to measure how full each substructure is (details can be found in [69].).

The shape indicator for a query path p in query Q can be calculated as $S_p = \frac{Size\ of\ element\ after\ shedding/spilling}{Size\ of\ element\ without\ shedding/spilling}$ [1].

When some substructures of a path are missing, its shape indicator is less than 1. Put differently, the quality of the path is "penalized " because of incomplete substructures.

In this work, the top most element is the smallest data unit based on which we can produce a result structure. We define *unit quality* as the quality gained by executing the reduced or the supplementary query on a top most element. We measure unit quality for a query using the formula below:

[1]Here we assume the size of an element is fixed.

$$\nu = \sum_{n}\sum_{p} N_p * S_p * \mu_(p) \tag{20.1}$$

Here n indicates the number of return structures generated per top most element. p denotes each substructure. N_p,S_p and μ_p denote the cardinality, shape indicator, and utility of p respectively.

Example 20.1.1. *Assume the input data stream for query Q1 as in Figure 20.2, now let us calculate the unit quality of the reduced and supplementary query for fusion candidate $[a^0, s^0, b^0, p^0, i^1, t^2, r^0]$. First consider the quality computation for the reduced query. Element i (i is short for ID) is spilled to disk. Two elements t (t for tel) are permanently dropped. In this case, the reduced output result is composed of only 1 substructure r (r for price). For substructure r, N is 2 since there are two r elements returned. Shape indicator S is 1 since complete r elements are output. μ is 1 since r is produced at runtime. Hence the reduced query quality is $2 * 1 * 1 = 2$. For the supplementary query, only element i is returned. Its age-based utility remains 1 even if its delivery is late. The quality of the supplementary query is 1 based on Equation 20.1 since only one i element is produced.*

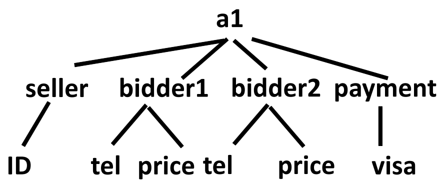


Figure 20.2: Data for Q1

20.2 Evaluating Costs for Reduced and Supplementary Queries

Our optimization goal is to optimize the total output quality for both the reduced and supplementary query. A cost model is used to measure the processing costs for both queries. We distinguish between two types of processing costs. One, the costs of processing a reduced query when the arrival rate is high, called *reduced cost*, and, two, the processing costs for supplementary queries when arrival rates are low, called *supplementary cost*. We first describe how to measure the costs for different processing operations, and then we put them together into a complete definition of reduced and supplementary costs.

In general, we measure the cost as the average time of processing a topmost element (we call it the unit processing cost). We divide the processing costs into the following parts: *Locating Cost* (LC) that measures the cost spent on retrieving data, *Join Cost* (JC) spent on structural joins and *Spill Cost* (FC) spent on flushing data.

Locating Cost. The locating cost indicates the cost spent on retrieving tokens. Automata are widely used for pattern retrieval over XML streams [22, 44]. The relevant tokens are "recognized" by the automata and then assembled into elements.

For both "shed" and "spill" paths, we need to locate the correct corresponding tokens. Therefore the locating costs are not saved when shed or spill is invoked. While the detailed locating cost model is discussed in [67], we estimate the locating cost savings using the formula below [67]:

$$C_L = \sum_{q \in A^{p_i}} n_{active}(q) C_{transit} \tag{20.2}$$

Here P_i indicates the query paths whose subtrees are contained by subtrees of shed or spilled paths. A^{p_i} denotes the set of states corresponding to P_i and its dependent states in

Notation	**Explanation**
A^{P_i}	Set of states of pattern P_i and its dependent states.
$n_{active}(q)$	The number of times that the stack top contains a state q when a start tag arrives
$C_{transit}$	Cost of transition to states in automaton
N_P	Number of elements matching P for a topmost element
$S_{\bowtie}$	Join selectivity
M_P	Size of P (number of tokens contained in each element)
C_j	Cost of comparing two elements
$C_{I/O}$	Cost of disk I/O
C_s	Cost of stack operation

Table 20.1: Notations Used in Cost Model

the automaton. $n_{active}(q)$ denotes state invoking times and $C_{transit}$ denotes the transition cost. The notations are in Table 20.1.

Join Cost. Since we assume stream data arrives in order, the elements for both join inputs are sorted. We can apply an efficient structural join algorithm, such as Stack-Tree-Anc [3], since both inputs are sorted. Using the cost model for this algorithm [70], we estimate the cost of structural join using the formula below :

$$C_J \ = \ 2 * N_V N_{B(i)} S_{\bowtie} C_j + 2 N_V C_s \tag{20.3}$$

Here N_V and $N_B(i)$ indicate the number of binding variables and branches per topmost element. Based on Equation 20.3, we can easily calculate the structural join savings for the reduced query. If a query path is marked as “shed” or “spill,” the structural join is not conducted at run time. Therefore for reduced queries, the join costs are saved for “shed” and “spill” paths.

Spill Costs. For "spill" paths, we consider the additional spill costs. As discussed in [69], we may have to flush other supporting paths to enable supplementary result generation. Let us use SP to denote the set of paths to be pushed to disk. The spill costs can be calculated as follows:

$$C_S = \sum_{p \in SP} N_p M_p C_{I/O} \tag{20.4}$$

For reduced query, since it may include the mixture of "keep", "shed" and "spill" paths, locating costs, join costs and spill costs need to be considered. The reduced query costs can be calculated using the formula below:

$$C(FC_i)^R = C_L + C_J + C_S$$

For the supplementary query, the spilled data is brought back from disk for query processing. In this case, the join costs need to be considered.

Chapter 21

Optimization Problem

21.1 Arrival Pattern

Prior work has focused on describing measuring arrival patterns in data stream [13, 24, 45, 60]. Most work either use statistics-based approaches or assume arrival patterns follow some known distributions. For instance, [24] and [13] assume input stream data arrival follows the exponential distribution. [45] assumes arrival rates follow uniform distribution for the input data. Other work [60] assumes arrival patterns can be estimated based on statistics. Here we utilize statistics to estimate the arrival pattern in the sense that we know the time period of the high arrival load and low arrival load. Without loss of generality, we assume the arrival pattern is a step-wise function. Now we focus on solving problems for the given arrival pattern–be it detected at runtime or formed a priori.

21.2 Formulation of Optimization Problem

Each fusion candidate FC_i in the FC lattice corresponds to a reduced and supplementary query pair. The reduced query performs “keep,” “spill” or “shed” actions on varying substructures and produces the reduced output. The supplementary query generates the supplementary results based on disk-resident data when system resources are available. Our quality and cost models introduced in Chapter 20.1 and Chapter 20.2 measure the unit quality and unit processing costs respectively. Decisions on the optimal reduced and supplementary query pair are affected by input data arrival patterns. Note that our quality model incorporates the age-based utility; therefore, the quality for supplementary queries is estimated based on by how much the supplementary results are delayed.

We aim to find an optimal fusion candidate that satisfies the following goals: the reduced query even when arrival rates are high must be able to keep up with the arrival rate, i.e., the reduced query is feasible, and the total quality including the quality for the reduced query and its matching supplementary query should be the highest among fusion candidates.

The fusion candidate selection problem is represented below.

Given the following inputs:

1. Varying arrival rates and their duration time periods:

$$< \lambda_1, t_1 >, \ldots < \lambda_i, t_i >, \ldots < \lambda_m, t_m >$$

Here λ_i denotes the arrival rate during time period t_i.

2. The unit computation costs of the reduced and supplementary query for each fusion candidate FC_i, denoted by $C(FC_i)^R$ and $C(FC_i)^S$ respectively.

3. Estimated output quality for the reduced and supplementary query:

For reduced query, since the processed data cannot exceed the incoming data, the output quality is calculated using the formula below:

$$Q(FC_i)^R \ = \ \nu_i^R * min\{\lambda, 1/C(FC_i)^R\}$$

Here ν_i^R denotes the unit quality of reduced query.

For supplementary query, its unit quality is affected by how much late the output is delivered. The output quality is calculated by:

$$Q(FC_i)^S \;=\; \nu_i^S * 1/C(FC_i)^S$$

Here ν_i^S denotes the unit quality of supplementary query.

Our goal is find a optimal fusion candidate to maximize the total output quality:

$$Max(\;Q(FC_i)^R + Q(FC_i)^S)$$

21.3 Algorithms for Optimizing Total Output Quality

The problem of choosing the optimal reduced and supplementary query pair can be transformed to choose the appropriate fusion candidate from the FC lattice which shed or spill data from input so that total output quality is the highest. We propose four optimization algorithms, OptF, OptSmart, HiX and FeX.

21.3.1 Optimal Fusion Search (OptF).

The baseline algorithm, that is guaranteed to return the optimal result, is to search the entire FC lattice and picks the fusion candidate that yields the highest total output quality. Here we call it Optimal Fusion Search (OptF), which is described in Algorithm 3.

Example 21.3.1. *Assume the arrival pattern is shown in Figure 21.1, with the fast arrival rate equal to 500 top most elements/s and the slow arrival rate 0 top most elements/s). In this case, the processing speed of the optimal reduced query needs to be faster than 500 top most elements/s. The data for Q1 is shown in Figure 21.2. The estimated unit processing costs and quality for each fusion candidate are shown in Figure 21.3. OptF searches the entire FC lattice and picks the fusion candidate* $[a^0, s^0, b^0, p^0, i^1, t^2, r^0]$ *as*

Algorithm 3 OptF Algorithm

```
1: // $FC_f$ indicates the optimal fusion candidate.
2: $FC_f = \emptyset$; $Q_{max} = 0$;
3: **for** each level $L_i$ in lattice **do**
4:     **for** each fusion candidate $FC_j$ in $L_i$ **do**
5:         **if** reduced query of $FC_j$ is feasible **and** $Q(FC_j) > Q_{max}$ **then**
6:             $FC_f = FC_j; Q_{max} = Q(FC_j)$;
7:         **end if**
8:     **end for**
9: **end for**
```

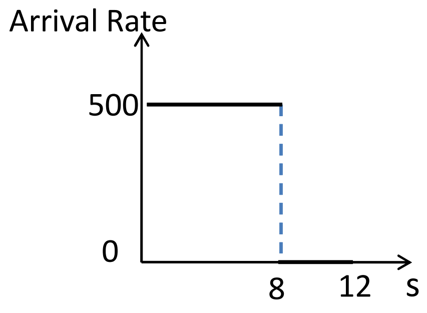


Figure 21.1: Arrival Pattern for Q1

optimal fusion candidate since its processing speed is higher than the arrival rate and its total output quality is the highest among all viable alternatives.

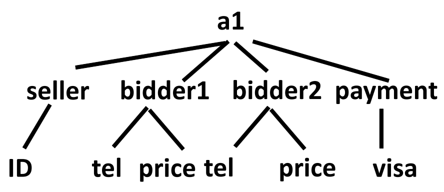


Figure 21.2: Data for Q1

Since OptF exhaustively traverses the search space, its search complexity is equal to the size of the candidate space, $O(3^{f^d})$, with fan-out f and depth of the query pattern tree d (details can be found in Chapter 19).

Fusion Candidate	Reduced Query		Supplementary Query		Quality /cost Ratio	Total Quality	Feasible ?
	Unit Quali ty	Unit Cost (ms)	Unit Quality	Unit Cost (ms)			
$[a^0,s^0,b^0,p^0,i^1,t^1,r^0]$	2	2.2	1+0.5*2	2.4	0.87	10600	N
$[a^0,s^0,b^0,p^0,i^2,t^0,r^0]$	4	2.8	0	0	1.43	11424	N
$[a^0,s^0,b^0,p^0,i^0,t^1,r^1]$	1	1.7	1+0.5*2	2.9	0.98	8000	Y
$[a^0,s^0,b^0,p^0,i^1,t^2,r^0]$	2	1.8	1	0.9	1.11	12000	Y
$[a^1,s^0,b^0,p^0,i^2,t^1,r^0]$	2	2.0	0.5*2	1.4	0.88	10856	Y
$[a^0,s^0,b^0,p^0,i^0,t^2,r^1]$	1	1.6	0	1.4	0.33	4000	Y
$[a^0,s^0,b^0,p^0,i^0,t^1,r^2]$	1	1.6	0.5*2	1.4	0.67	6856	Y

Figure 21.3: Quality and Cost of Fusion Candiates

21.3.2 Optimal Search with Smart Pruning (OptSmart).

Since OptF needs to search the entire FC lattice, the complexity of OptF is high. We now design the Optimal Search with Smart Pruning approach (OptSmart) that applies pruning to eliminate suboptimal solutions. OptSmart succeeds in improving the efficiency of the search cost without compromising the result optimality. OptSmart is described in Algorithm 4.

Algorithm 4 OptSmart Algorithm

```
1: // $FC_f$ indicates the optimal fusion candidate.
2: $FC_f = \emptyset$; $Q_{max} = 0$;
3: for each level $L_i$ in lattice do
4:     for each fusion candidate $FC_j$ in $L_i$ do
5:         if reduced query of $FC_j$ is feasible then
6:             if $Q(FC_j) > Q_{max}$ then
7:                 $FC_f = FC_j; Q_{max} = Q(FC_j)$;
8:             end if
9:             Prune all descendants of $FC_j$ on lattice;
10:        end if
11:    end for
12: end for
```

OptSmart guarantees to find the optimal fusion candidate based on the following observation.

Observation 21.3.1. *In the top-down FC lattice traversal, when we reach a candidate FC_i and find its reduced query is a feasible query, the quality of its descendants is guaranteed to be not higher than that of FC_i.*

Proof. Assume fusion candidate FC_i produces r_i result structures. Now let us compare candidate FC_i and its children on level $i + 1$. First consider reduced queries. Recall that a child candidate either changes the action of some substructure from "keep" to "spill" or from "spill" to "shed." Thus the quality of each reduced output result structure for a child candidate cannot be higher than that of candidate FC_i. Furthermore, the number of output result structures cannot increase since all input data is consumed. So the reduced query quality of child candidates cannot be higher than that of the initial parent candidate FC_i.

Now let us consider the quality of the supplementary queries. When the action on some substructure for a child candidate is changed from "keep" to "spill," then this implies the supplementary query quality for those substructures is degraded due to delay. If the action on some structure is changed from "spill" to "shed," the supplementary query quality of a child candidate can never increase since no data for that substructure will be brought back later. Therefore, the total output quality of a child candidate of FC_i is guaranteed to be not higher than that of FC_i. Similarly, we can prove that the quality of descendants of FC_i is guaranteed to be not higher than that of FC_i. □

Example 21.3.2. *In Figure 21.4, the reduced query of candidate $[a^0, s^0, b^0, p^0, i^0, t^1, r^1]$ on level 3 can keep up with the arrival speed. In this case, we can prune its children $[a^0, s^0, b^0, p^0, i^0, t^2, r^1]$ and $[a^0, s^0, b^0, p^0, i^0, t^1, r^2]$ and other descendants. Similarly, $[a^0, s^0, b^0, p^0, i^2, t^1, r^0]$ and $[a^0, s^0, b^0, p^0, i^1, t^2, r^0]$ can consume all the input. Thus their* *descendants can be pruned.*

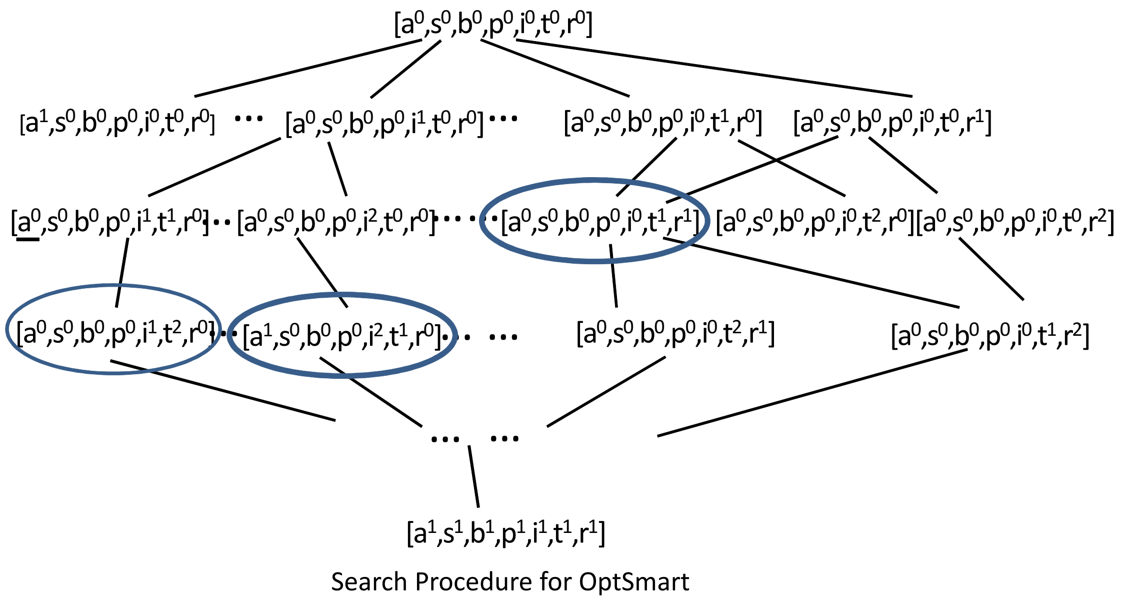


Figure 21.4: OptSmart Search Example

To estimate the search complexity, the worse case for OptSmart is to check every candidate without pruning anything. Therefore the complexity of OptSmart is the same as that of OptF, which is $O(3^{f^d})$. However, our experimental results will show that the complexity of OptSmart is much smaller than OptF.

21.3.3 Hill-climbing Heuristics (HiX).

We now present a Hill-climbing Heuristic (HiX), which has much more efficient running time compared to OptF and OptSmart. The heuristic is based on the conviction that the candidate with highest quality/cost ratio should yield the highest output quality. The algorithm is described as below.

Example 21.3.3. *The HiX search for query Q1 is shown in Figure 21.5. On level 2, the quality/cost ratio of the fusion candidate* $[a^0, s^0, b^0, p^0, i^1, t^0, r^0]$ *is the highest. So*

Algorithm 5 HiX Algorithm

```
1: // $FC_f$ indicates the optimal fusion candidate.
2: $FC_f$ =root; $Q_{max} = Q_{root}$; $L_i = 1$; // $L_i$ is the current level
3: while true do
4:     Check child candidates of $FC_f$ on next level;
5:     Pick child candidate $FC_f$ with highest quality/cost ratio;
6:     if reduced query of $FC_f$ is feasible then
7:         Stop;
8:     else
9:         $L_i = L_i + 1$; //Move to the next level
10:    end if
11: end while
```

we only explore its children on level 3. On level 3, the quality/cost ratio of candidate $[a^0, s^0, b^0, p^0, i^2, t^0, r^0]$ *is the highest. We finally stop at candidate* $[a^0, s^0, b^0, p^0, i^2, t^1, r^0]$ *since it is feasible. However,* $[a^0, s^0, b^0, p^0, i^2, t^1, r^0]$ *is not the optimal candidate for this problem since as we know, the total quality of* $[a^0, s^0, b^0, p^0, i^1, t^2, r^0]$ *is the highest.*

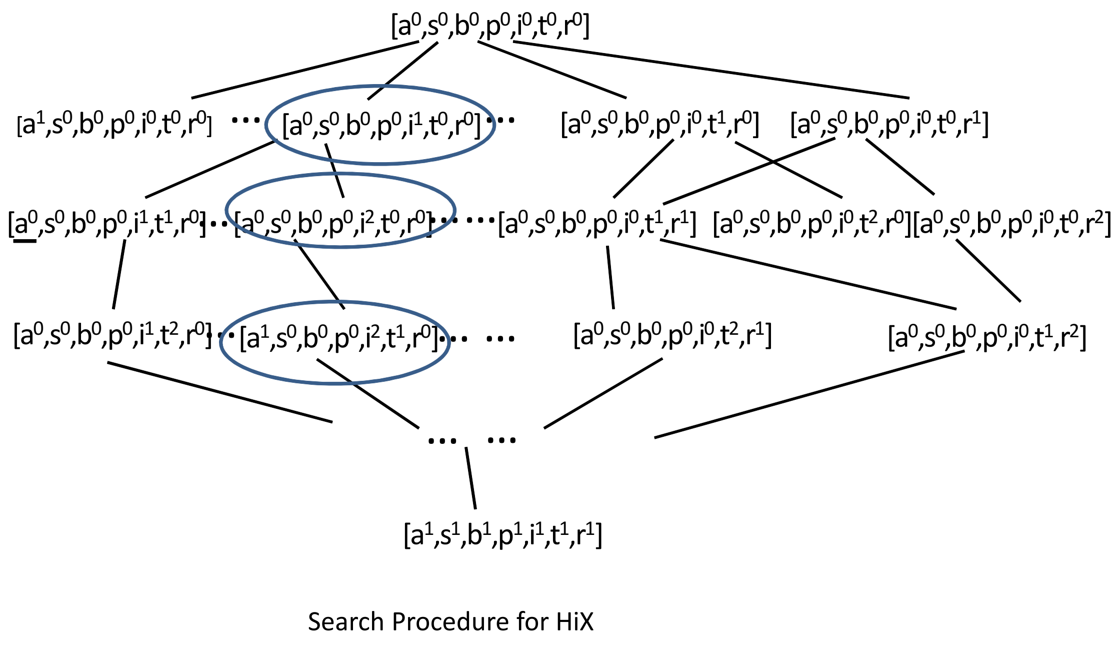


Figure 21.5: HiX Search Example

Complexity analysis. As discussed in Chapter 19.3, every fusion candidate has at most τ children in the lattice (τ indicates the total number of nodes in query tree). Hence we check at most τ candidates on each level. The lattice has totally 2τ levels. Thus in the worse case we have to run this search 2τ times. So the total search cost is τ^2, which is bounded by $O(f^{2d})$. The complexity of HiX is much smaller than that of OptF and OptSmart.

HiX may end up finding a locally optimal fusion candidate. The reason is when it explores the candidates from top to down, it only checks the child candidates of the fusion candidate with highest quality/cost ratio. The neighbors of the candidate with highest quality/cost ratio are skipped. Therefore, it is not guaranteed to return the globally optimal candidate. However, HiX is more efficient than OptF and OptSmart.

21.3.4 Fast EXplore Heuristics (FeX).

When a query is very complex, HiX may still be a costly search. We design a Fast EXplore heuristic (FeX), which is even more efficient than the above approaches. FeX randomly picks a fusion candidate on each level in a top-down manner until finding a feasible candidate.

Algorithm 6 FeX Algorithm

```
1: // $FC_f$ indicates the optimal fusion candidate.
2: $FC_f$ =root; $Q_{max} = Q_{root}$; $L_i = 1$; // $L_i$ is the current level
3: while true do
4:     Randomly pick a candidate $FC_f$ on next level;
5:     if reduced query of $FC_f$ is feasible then
6:         Stop;
7:     else
8:         $L_i = L_i + 1$; //Move to the next level
9:     end if
10: end while
```

We ran the above algorithm K times, finally we pick the one with highest total output quality.

The complexity of FeX is decided by the iteration input control K and the number of levels in the FC lattice that is bounded by $O(f^d K)$. Although there is no optimal candidate guarantee for FeX, the complexity of FeX is much smaller than that of OptF and OptSmart.

21.4 Supplementary Query Execution Policy

When input data arrival slows down and thus CPU resources remain available, then we can proceed to bring the data on the disk back to execute the corresponding supplementary query. Note that we do not have the load overflow problem for supplementary queries due to the slow arrival speed of the input data. The data is read from disk in a pull-based manner, unlike in the push-based stream case on when reduced query operate. We now note that over time, the optimization algorithms may have been triggered multiple times. This means several alternate pairs of optimal reduced queries and supplementary queries may have been chosen over time. To avoid the old disk-resident to expire leading to quality loss, we employ a freshness-based supplementary query execution policy. The data spilled to disk is brought back for processing based on their spill time order. In other words, through shipping any historical spilled data which may have become so stale that its quality is estimated to be equal to zero now, the optimal supplementary query generated earlier will be executed first.

Chapter 22

Experimental Results

We conducted extensive experiments to compare four optimization algorithms OptF, OptSmart, HiX and FeX. We also employed an algorithm, called *Random*, which iteratively selects one among all fusion candidates randomly until enough substructures are dropped or spilled so that the input load can be handled by the corresponding reduced query. We first compare the performance of our optimization algorithms with the Random approach. The experimental results demonstrate that our optimization algorithms consistently achieve higher quality than the Random approach. In addition, we compared our optimization approaches which generated optimal fusion candidates with pure shed and pure spill optimization approaches. The experimental results demonstrate that our integrated framework has better performance over the pure shed and pure spill approaches. We performed the following four sets of experiments:

- The first set of experiments compared the performance of our optimization algorithms with Random approach.
- The second set of experiments compared the performance of our optimization approaches with pure shed and pure spill optimization approaches.

- The third set of experiments compared the performance of our optimization algorithms for various selectivity settings.
- The fourth set of experiments examined the overhead of different optimization algorithms.

Experimental Setup. We have implemented our proposed optimization approaches in an XML stream system called Raindrop [30]. We use ToXgene [12] to generate our testing data. All experiments are run on a 2.4GHz i3 processor with 4096MB memory.

22.1 Comparison of Our Optimization Approaches

The first set of experiments compared the performance of our optimization algorithms with Random approach in two cases. One case is fast and reliable network. The other case is the network that is unreliable, i.e., the arrival pattern shows a mixture of fast arrival rates and slow arrival rates.

22.1.1 Reliable Networks

When the network is reliable, the network never incurs suspensions of data transmission. In this set of experiments, we set arrival intervals between two top most elements 0.03s. The arrival rate was higher than the processing speed. In this case, the supplementary query never had a chance to be invoked. We used Q1 as the running query. Optimization was invoked as soon as the memory buffer threshold was reached. We measured the cumulative output quality gained by using varying optimization approaches. Figures 22.1 shows the cumulative output quality using four optimization strategies when the arrival interval is 0.03s.

We observed that OptF, OptSmart, HiX, and FeX gain higher total quality than the Random approach. In addition, OptF and OptSmart both gained much higher quality than HiX, FeX and Random. This is because OptF and OptSmart were designed to find the optimal fusion candidate. Since the arrival speed was higher than the processing speed, supplementary query was not invoked. In this case, OptF and OptSmart choose the optimal reduced query to achieve highest output quality.

22.1.2 Unreliable Networks

Now let us examine the performance of the optimization approaches in the scenario of unreliable networks. To simulate unreliable network, we generated arrival intervals using Pareto distribution that is widely used in case of a bursty network [20]. The cumulative quality for our optimization approaches is shown in Figure 22.2. Observe that Figure 22.2 shows step-like performance for all the optimization approaches due to switching between the reduced query and the supplementary query. This is because when no data arrives, supplementary query gets a chance to be executed. In addition, the slope of segments corresponding to the spilling phase for OptF and OptSmart is larger than that of HiX, FeX and Random. This indicates that output quality for OptF and OptSmart is increased faster than that of HiX, FeX and Random.

22.2 Comparison of Our Approach with Pure Shed and Spill Approaches

The second set of experiments compared our optimization approach with state-of-the-art pure shed and spill optimization approaches. [67] proposed structure-based shedding approaches to selectively drop substructures to permanently reduce workload. We call this purely shedding approach that chooses the optimal shed candidate *P-Shed*. [68] proposed a structure-based spilling approach that selectively flushes less time-critical XML

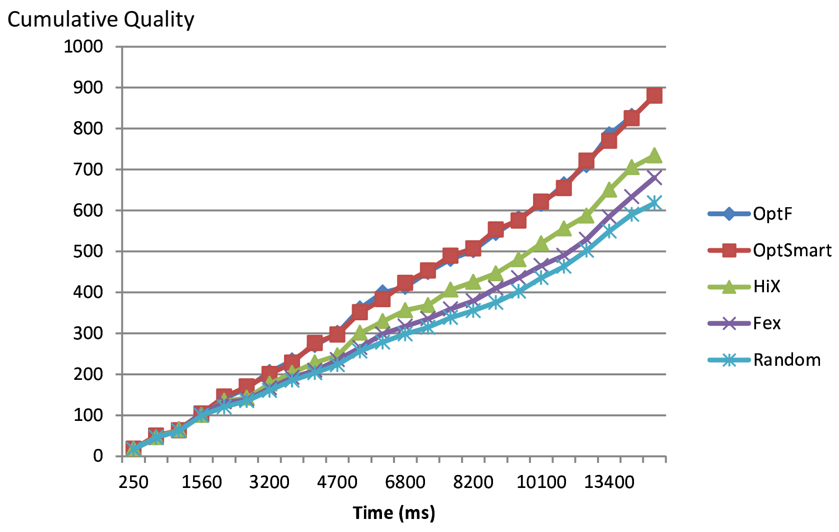


Figure 22.1: Reliable Network

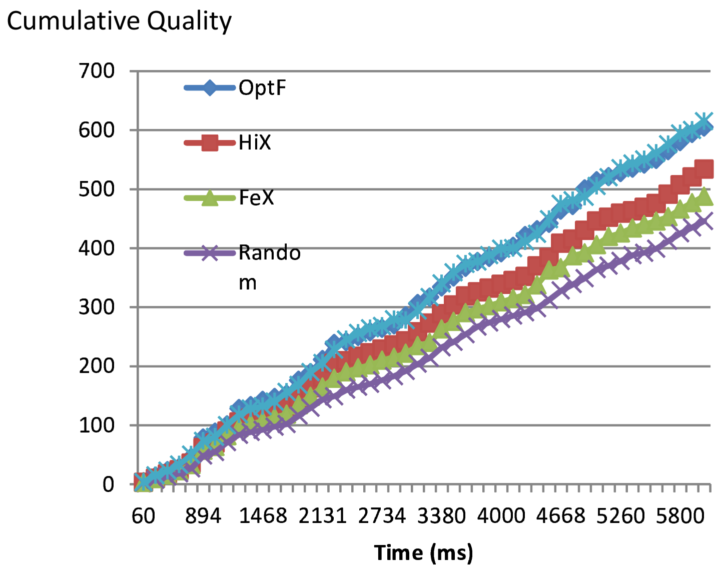
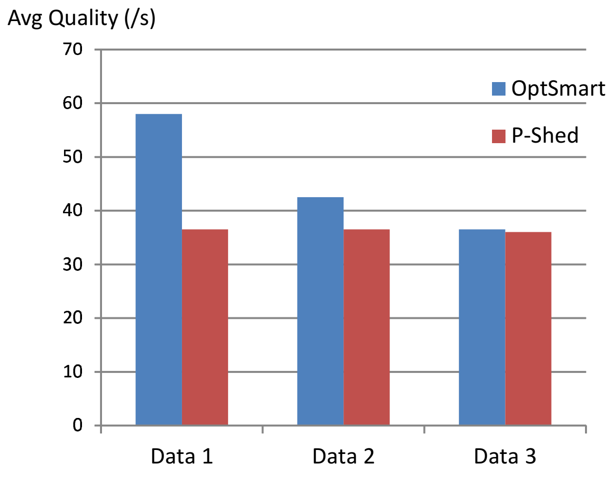


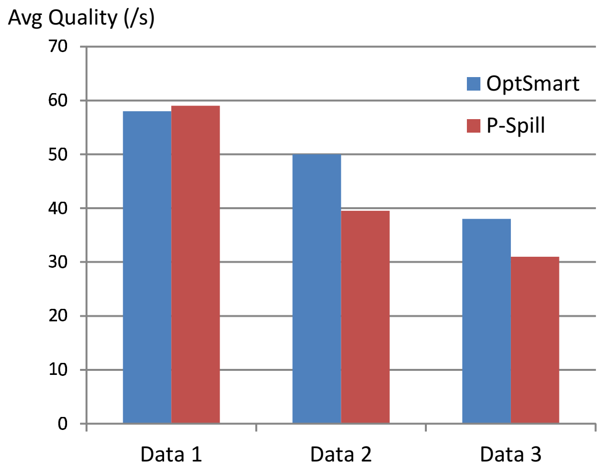
Figure 22.2: Unreliable Network

substructures to disk. We call this pure spill approach that produces the optimal spill candidate *P-Spill*.

In this set of experiments, the arrival intervals were generated using Pareto distribution to simulate fluctuating arrival pattern. Since P-Shed and P-Spill both generate optimal candidates, we compared them with OptSmart which also guarantees to generate the optimal fusion candidate.



(a) Our Approach vs. P-Shed



(b) Our Approach vs. P-Spill

Figure 22.3: Performance Comparison of Our Approach with P-Shed and P-Spill

Let us fist examine the performance comparison between P-Shed and our approach. We generated three data sets that vary on their age-based quality. For data set 1, the quality of all query paths remained unchanged when output was delayed. For data set 2, the quality of all query paths dropped to 1/2 of their initial quality when delayed. For data set 3, the quality of all query paths dropped to 0 if delayed. Figure 22.3(a) shows the quality comparison of P-Shed and OptSmart. Observe that for data set 3, when the quality of all query paths drops to 0 if delayed, OptSmart and P-Shed had the same output quality since they both pick the shed candidate to achieve the highest output quality. For data sets 1 and 2, the quality of OptSmart was higher than that of P-Shed. The reason is OptSmart chooses the optimal fusion candidate that is producing delayed output based on disk-resident data even when no data arrives while P-Shed always permanently drops data and produces nothing when no data arrives.

To compare the performance of P-Spill with OptSmart, we generated the following three data sets. For data set 1, the quality of all query paths remained unchanged. For data set 2, the quality of 50% query paths dropped to 1/2 of their initial quality if delayed. Quality of the other 50% query paths remained unchanged. For data set 3, the quality of all query paths dropped to 0 when delayed. Figure 22.3(b) shows the quality comparison of P-Spill and OptSmart. Observe for data set 1, the quality of OptSmart and P-Spill was the same. Since the quality of all the query paths remains unchanged if delayed, OptSmart and P-Spill both choose to flush less time-critical data to disk to achieve the highest output quality. For data set 2, our approach wins over P-Spill. The reason is to make the reduced query fast enough to keep up with input arrival rate, OptSmart selectively drops the query paths whose quality degrades quickly while spilling the query paths whose quality is unchanged. However, the P-Spill is limited to always spilling data to disk to reduce the workload. The P-Spill thus had higher reduced query costs than our approaches due to always spilling data. For data set 3, the quality of OptSmart was also higher than P-Spill because of the same reason.

22.3 Impact of Selectivity

In this set of experiments, we illustrate that the output quality was affected by the selectivity distribution of the binding variable and each branch.

The test data sets satisfy the following requirements: 1) each returned query path contained the same number of tokens and corresponded to the same numbers of elements, and 2) the element sizes corresponding to each returned path were equal. Based on the cost model in Chapter 20.2, the locating and join costs spent on each returned path are the same. We used five different sets of selectivity that differ in their standard deviations. Figure 22.4 shows that the output quality was higher when there was a bigger variance among selectivity for OptF and OptSmart. This is because OptF and OptSmart choose to shed or spill the return paths with low selectivity which yield low output quality given the same computation cost. However, the quality of the reduced query achieved by the Random approach did not change much because Random approach did not keep the returned paths with large selectivity.

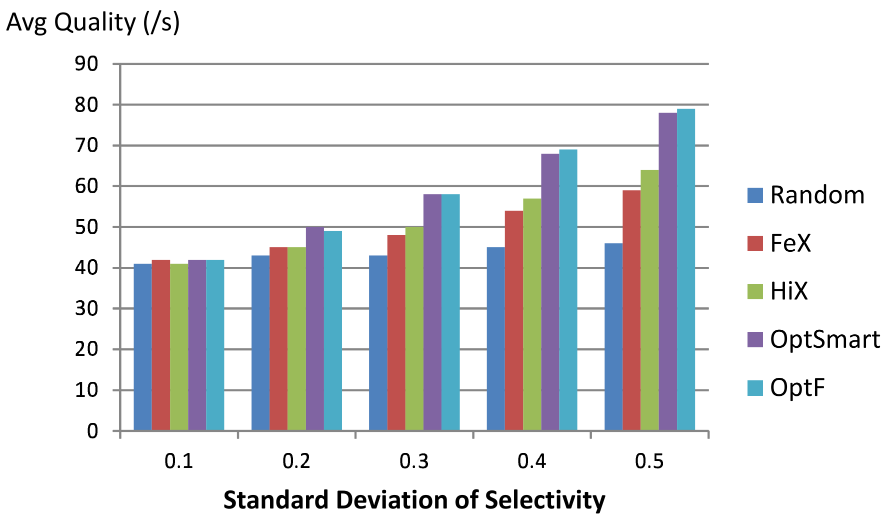


Figure 22.4: Quality for Varying Selectivity

22.4 Overhead of Optimization Approaches

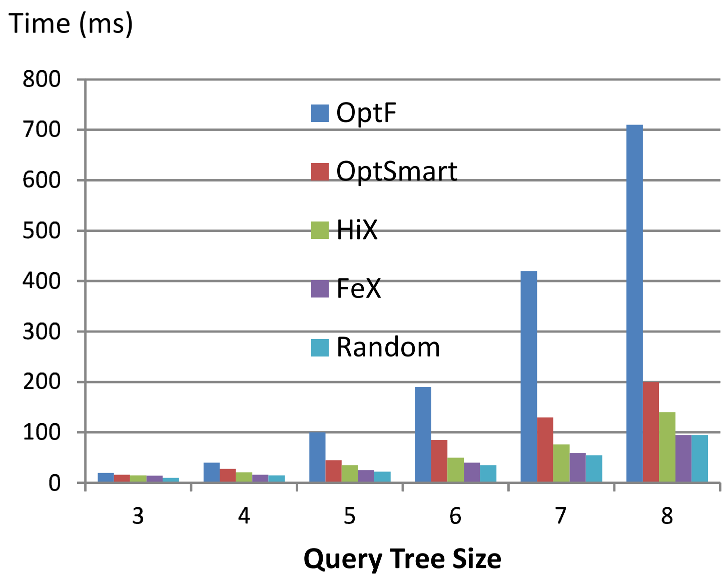


Figure 22.5: Overhead of Optimization Approaches

The fourth set of experiments examined the overhead of our optimization approaches. The overhead of our optimization strategies was measured by the time spent on choosing the optimal fusion candidate. We examine the relationship between the complexity of the query and the overhead of the optimization methods. In this set of experiments, we used five queries which varied in the size of the query trees. Figure 22.5 shows the overhead of optimization approaches. Note that the overhead of FeX remained low when the query became more complex since it checks, at most, one fusion candidate on each level of FC lattice. In Figure 22.5, when the queries become complex, the overhead of OptF was much higher than that of other approaches since it was always searching the optimal fusion candidate with the cost of the entire FC lattice. The overhead of OptSmart was much smaller than that of OptF. This indicates that our pruning method is indeed effective at reducing the search cost.

Chapter 23

Related Work

The current state-of-the-art in load shedding for relational stream systems can be categorized into two main approaches [10, 21, 28, 59]. One is random load shedding [59], where a certain percentage of randomly selected tuples is discarded. The other approach is semantic load shedding which assigns priorities to tuples based on their utility to the output application and then sheds those with low priority. Our shedding approach can be regarded as semantic shedding, but on structural data. This means shedding objects are not whole tuples but rather substructures. We assign priorities to substructures instead of tuple values.

Preference model is widely used for decision making purposes in many applications, such as e-commerce and personalized web services. Aurora [59] combines the utility of different tuple values into quality of service. [38] proposes Preference SQL, an extension of SQL that is able to support user-definable preferences for personalized search engines. Preference XPath [37] provides a language to help users in E-commerce to express explicit preferences in the form of XPath queries. We can use their language to express the preferences of different substructure in the query.

Spilling techniques have been investigated in relational streams. Flush algorithms have been proposed to either maximize the output rate or to generate a subset of result set as early as possible [39, 42, 43, 48, 58, 62, 63]. However, we cannot directly apply their techniques into structure-based spilling in XML streams because of the following reasons: 1) the spilled objects in relational streams are tuples. However, in our context, spilled objects are substructures of the hierarchical XML data, and 2) these works are focusing on providing non-blocking flush techniques when conducting a different relational join, such as Symmetric Hash Join, Hash-Merge Join and Progressive Merge Join. However, structural join is the core component of XQuery plans, which can be looked as a θ join whose condition is to compare the regions of two elements [71].

[54] first proposes to produce approximate results for XQuery when no input for some operators in the plan exists. However, they do not address the case that substructures are missing from the input. In addition, since they assume the data is persistent, supplementary query result generation does not require spilling extra data.

My earlier work on structural shedding [67], as presented in Part I of this dissertation, is the first to deal with the problem of selectively dropping XML subelements to achieve high processing speed. [67] assumes the returned query patterns are independent from each other. Hence the data dependency issue among varying query patterns is not addressed. [69] tackles the problem of selectively choosing substructures to spill to disk and generating complete output. [69] addresses the issue of producing runtime output by determining the correct spilling effect in query due to data dependency among varying query patterns. In this work, we focus on examining fusion candidates which is the hybrid of structural shedding and spilling. We propose a carefully calibrated multi-method load reduction framework that applies both structural shedding and spilling technology to achieve maximal effectiveness in processing input streams.

Part IV

Conclusions and Future Work

Chapter 24

Conclusions of This Dissertation

24.1 Summary of Dissertation

Stream applications are often characterized by push-based data sources in which the arrival rates can be high and unpredictable. When the arrival rate is very high, stream processing systems may not always be able to keep up with the input data streams. In this dissertation, two load reduction techniques, including structural shedding and spilling techniques, were proposed for XML stream processing to solve the issue of insufficient system resources to keep up with the processing of the stream.

In the first part of this dissertation, we focused on the problem of structural shedding for XML streams. We proposed a new utility-driven load shedding strategy that exploits features specific to XML stream processing. Our preference model for XQuery helped users to customize their preferences on different XML result structures. We designed a cost model for estimating the costs of different shed queries. The shedding problem was formulated as an optimization problem, namely, to find the appropriate shed queries to maximize the output utility. To solve the shedding problem, two shed query searching solutions, OptShed and FastShed, were proposed to choose a subset of shed queries to be executed in order to maximize utility. OptShed guaranteed to find an optimal solution however at the cost of an exponential complexity. FastShed achieves a close-to-optimal result in a wide range of cases with much smaller search costs than OptShed. In addition, a simple yet elegant in-automaton shedding mechanism was proposed by suspending the appropriate states in the automaton-based execution engine for XML streams, in order to drop data early.

In the second part of this dissertation, we focused on the problem of structural spilling for XML streams. We proposed the first structure-based spilling strategy that exploits features specific to XML stream processing. Our structure-based spilling framework was general and can be applied in any XML stream system. We analyzed the effect on different paths in query for a particular spilled path. How to execute reduced queries given varying spilling effects on the query was examined. An output quality model was proposed for evaluating the quality of partial returned structures. We proposed a cost model for measuring the execution cost for different reduced queries. In addition, to eventually produce entire yet duplicate-free result set, an output model was proposed to match supplementary “delta” structures with partial result structures produced earlier. To generate supplementary results, we determined what extra data to spill to disk to guarantee that the entire result set can be produced. To solve the spilling problem, we developed three strategies, OptR, OptPrune and ToX. OptR and OptPrune were guaranteed to find the optimal structures to spill. ToX cannot guarantee to find the optimal structures to spill. When the queries became complex, the overhead of ToX was much smaller than OptPrune and OptR since it stopped at the earliest candidate which consumes all input. We could use OptPrune approach when the query is not very complex since its pruning method is indeed effective at reducing the search cost. However, when the query became more complex, OptPrune may not be a practical solution since its overhead was larger than ToX. In this case, we resolved to utilize our lightweight ToX solution.

In Part I and Part II we discussed the structural shedding and structural spilling techniques for XML streams. However, in some scenarios, critical disadvantages exist for both the shed as well as the spill techniques. On the one hand, shedding data means that partial output is lost forever. In addition, dropped data may lead to blocked output, especially when there is a lull in the input. On the other hand, spilling makes the strong assumption that system resources will be ample to process all disk-resident data sooner or later. In the third part of dissertation work, we proposed a novel integrated framework for a hybrid structure-based shed and spill approach which is able to be applied in any XML stream system. The structure-based shedding and spilling problem was formulated into an optimization problem, namely, to find a pair of the reduced and supplementary queries that maximizes the output quality. We designed a solution space for fusion candidates that represents all the shed and spill candidates. An age-based quality model was proposed for evaluating the output quality for different reduced and supplementary query pairs. A family of four optimization strategies, OptF, OptSmart, HiX and Fex, were proposed to find the optimal fusion candidate which maximizes the total output quality. Our experimental results demonstrate that our proposed solutions consistently achieved higher quality results compared to the state-of-the-art techniques.

24.2 Discussion of Three Parts

Part I and Part II of this dissertation explore the problems of structural shedding and structural spilling for XML streams respectively. Although structural shedding and spilling look similar in the sense of reducing the workload when the arrival rates are high, they vary in their assumption, quality measurement, and reduced candidate representation.

For structural shedding problem, we assumed that returned paths in a query were independent from one another. However, for structural spilling problems, this limitation was removed. Hence for structural spilling, the spilling side effects on the query caused by pushing a single query path to disk were examined in Part II. For structural shedding problem, we allowed users to assign preference to varying query paths. Two types of preference settings, namely, prioritized and numerical preference models, were utilized to represent the importance of query paths. In addition, a scoring scheme for patterns without preferences was proposed in the preference model. However, for structural spilling, prioritized and numerical preference models cannot be directly applied because the assumption that return paths were independent from one another does not hold for this problem.

Structural shedding employs a reduced candidate representation strategy different from structural spilling problem. For structural shedding problem, we used shed queries, the queries generated by removing one or more nodes from the query pattern tree for a given query, to represent possible reduced candidates. We aimed to find a set of shed queries to optimize output quality. As we mentioned earlier, for structural spilling problem, we must consider the spilling side effects caused by spilling a query path. In this case, a minimum candidate space was proposed to avoid unnecessary investigation on reduced queries resulting in the same spilling side effects.

In the integrated framework for structure-based shed and spill, we focused on investigating fusion candidates that is the hybrid of structural shedding and spilling. For this integrated framework, we cannot just plug in our methodologies from part I and part II. Structural shedding solutions simply assume the delayed output was no longer needed, thus a pure shedding approach would be sufficient. For structural spilling solution, a clean up stage, which triggers supplementary query execution to produce supplementary results to complements output generated earlier, was guaranteed when the system has enough resources. For an integrated framework supporting the hybrid of shed and spill, we needed a means to measure how much output quality was affected if output is delayed.

Hence a new age-based utility model was proposed for the integrated framework.

Chapter 25

Future Work

This dissertation explores the structural shedding and spilling in XML streams. There are many open unsolved research challenges in this area. This chapter discusses several future work topics that are important for load reduction techniques in XML streams. In particular, the topics for future work include: 1) Combining automaton-in-out query optimization with structural shedding/spilling, 2) Multi-query shedding/spilling in XML streams, 3) Supporting hybrid preference model in load shedding/spilling, 4) Organizing of indexing flushed data on disk, and 5) Load spilling for XQuery with window functions.

25.1 Combining Automaton-in-out Query Optimization with Structural Shedding/Spilling

In this dissertation, the query plan generation follows the following rule: all query patterns are retrieved in the automaton. Then the collected data is passed up to the algebra plan. The structural shedding/spilling algorithms mainly focus on choosing substructures to drop or to flush so as to maximize the output quality. When the query processing rate cannot keep up with the input data arrival rate, shedding/spilling is invoked. However, the query plan shape may affect query processing costs. When the query processing rate cannot keep up with the arrival rate, switching to another plan may be able to keep up with the arrival rate. In other words, query optimization may generate a plan with lower computation costs than the initial plan, which can keep up with the arrival rate. If this is the case, shedding or spilling is not necessary. Let me illustrate this via an example query as follows:

```
Q3: FOR $a in stream()/auctions/auction[reserve]
  WHERE $a//profile contains “frequent”
  RETURN <auction> $a/seller, $a/bidder </auction>
```

The corresponding automaton and algebra plan for Q3 are shown in Figure 25.1. In Figure 25.1, pattern <auction> is retrieved by the automaton. In other words, when a start tag <auction> is encountered, we start collecting tokens. We stop collecting when an end tag </auction> is encountered. The collected tokens are further passed up to the algebra plan on the top.

Clearly, in Figure 25.1, only one pattern $/auctions/auction$ is retrieved in the automaton. Other patterns in the query, such as $\$a/reserve$ and $\$a//profile$, are obtained by navigating into $auction$ elements. However, this may not be an optimal plan. Figure 25.2 shows another plan. In this plan, we push pattern retrieval on $reserve$ into the automaton. In this case, only the $auction$ elements that have $reserve$ children will be passed up to the plan on the top. When very few $auction$ elements have $reserve$ children, the plan in Figure 25.2 results in lower computation costs compared with the plan in Figure 25.1. Based on the above observation, changing the retrieval of a pattern by placing them inside or out of the automaton may affect the plan costs. We call it *automaton in-out optimization* [57]. Since each pattern in the query can be retrieved in or out of automaton, alternative plans are generated by pulling the pattern retrieval out of automa-

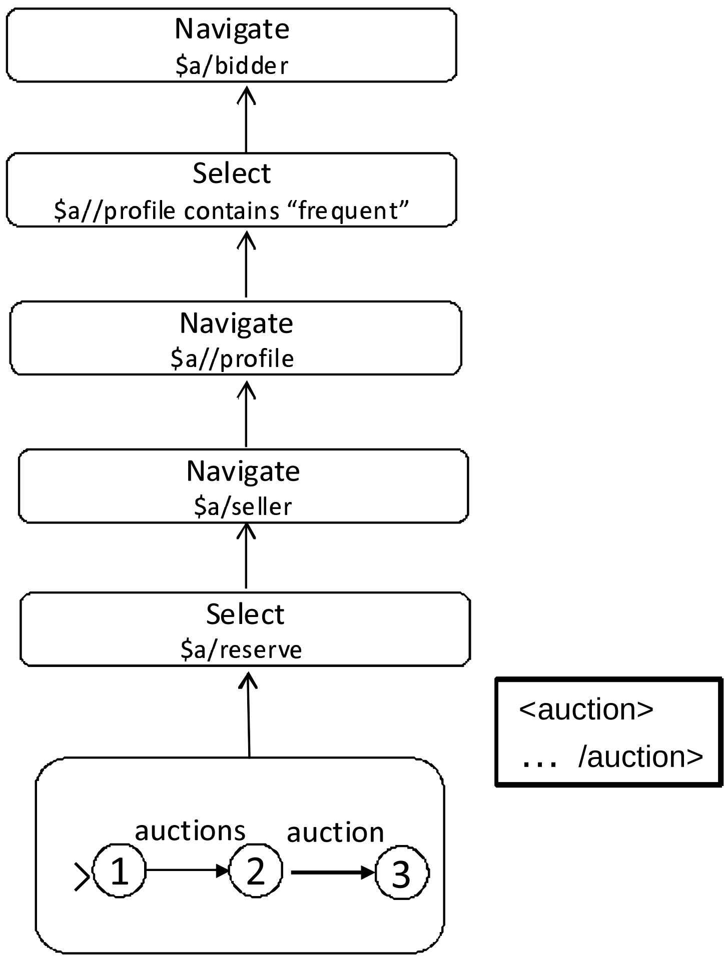


Figure 25.1: Plan for Q3

ton or pushing the pattern retrieval into the automaton. [57] describes the rewriting rule that could be employed to produce these alternative plans by rewriting one plan into an alternative one. We can examine these alternative query plans and find an optimal plan with the lowest cost.

Query optimization can improve the query processing speed, while keeping the output accuracy. In addition, load shedding/spilling has to be invoked when the optimal plan is not fast enough to keep up with the arrival rate. Therefore, we need to consider the

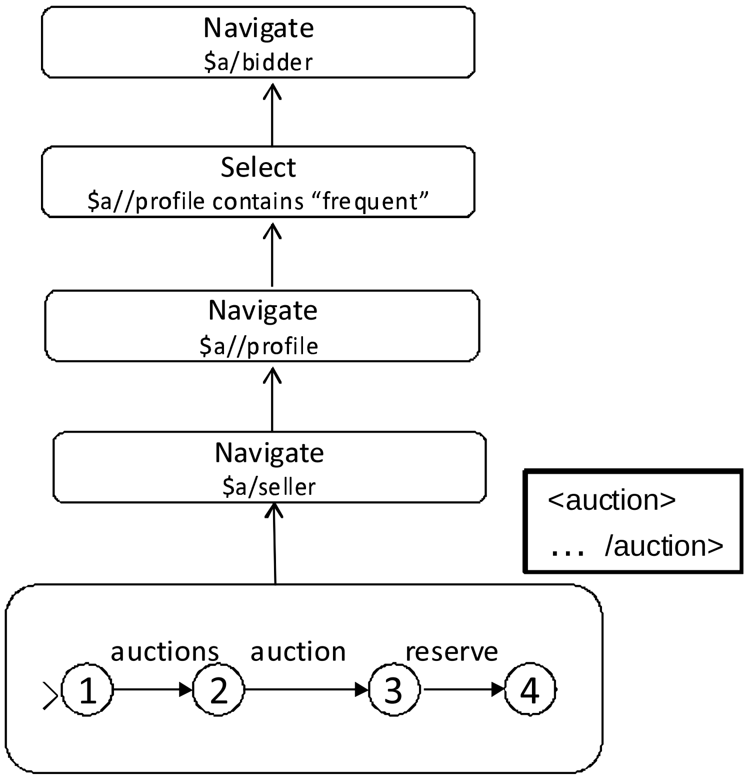


Figure 25.2: Alternative Plan for Q3

interrelationship between automaton-in-out optimization and structural shedding/spilling. We can combine the automaton-in-out optimization with load shedding/spilling into a new optimization problem. The goal is to find a plan which can consume all the input data and produce as many output results as possible. To accomplish this, we need to tackle the following challenges. First, we need to determine whether the optimal plan generated by the query optimizer can consume all the input data, i.e., the query processing rate of the optimal plan can keep up with the arrival rate. Two, we need to estimate the cost of finding an optimal plan. If the cost of finding an optimal plan is too high and the optimal plan cannot keep up with the arrival speed, we have to switch to a reduced plan which drops data from the input or flushes data to disk temporarily.

25.2 Multi-Query Load Shedding and Spilling

In this dissertation, we focus on the structural shedding and spilling for a single query. In the future work, one could explore structural shedding spilling for multiple queries. Multi-query load shedding and spilling bring up new challenges. First, since multiple queries may need to extract common XQuery expressions, an efficient query execution paradigm which employs a shared processing approach must be designed. As discussed earlier, query patterns are retrieved in the automaton. This requires us to design an automaton for multiple queries carefully so that duplicate transition in pattern retrieval can be avoided. In addition, the changes on the automaton when adding a new query and removing an existing query must be simple so that these can be conducted online. Second, since queries may be submitted by different users, the query preference settings may vary. How to choose the substructures to drop or spill to maximize the total output quality for multiple queries is an important issue. Finally, since we propose the solution for structural shedding/spilling on a single query in this dissertation, whether we can apply current structural shedding/spilling solutions to multi-query workload when faced with insufficient main memory and CPU processing resources is an interesting problem.

25.3 Supporting Hybrid Preference Model in Load Shedding/Spilling

In the future work, one could employ a hybrid preference which combines both structure-based and value-based preference in load shedding/spilling techniques.

In the XML stream scenario, the input stream as well as the output result are composed of different XML subelements instead of just flat attributes, and hence more complex than relational tuples. As we discussed in Chapter 4, we propose a structure-based preference model for XML stream. In a structure-based preference model, the importance of different elements in an XML tree may vary due to their semantics. However, this structure-based preference model does not look at the values for a substructure, which may also affect the output quality in practical applications. Consider a social network website scenario. Users may edit their personal profiles and send messages to their friends at any time. Status updates, composed of possibly nested structures including updates from friends, recent posts on the wall and news from the subscribed group, are generated continuously. Different users may be interested in specific primary updates. For instance, a college student wants to make new friends in Boston area. He wants to be notified when his friends add new friends. When the system resources are limited, it may be favorable to delay the output of unimportant updates and instead only report "favorite updates" to the end users. In this case, the "favorite" substructure for this user is "friend". In addition, the "favorite" value of a new friend's location is "Boston" since he is interested in new friends in Boston area. In this case, a hybrid preference model composed of both structure-based and value-based preferences is needed to report "favorite updates" to the end users when system resources are limited.

To support hybrid preference, a new means to represent both structure-based and value-based preference for XML data needs to be explored. A quality model for evaluating the output quality based on the hybrid preference model needs to be addressed. In addition, as discussed in Chapter 6, many XML stream systems use an automaton to recognize relevant elements on incoming data streams. Since dropping input data as early as possible can avoid wasted work, the unimportant substructures can be dropped when we recognize the corresponding tokens using automaton. Similarly, for a hybrid preference model, we need to drop the unimportant substructures with unimportant values as early as possible. When detecting a substructure to drop using an automaton, how to add a value-based filter on this substructure is an important issue.

25.4 Organizing of Indexing Flushed Data on Disk

When spilling data to disk, the data should be organized in a fashion way so to facilitate the processing of the supplementary query. In this dissertation, we take a simple approach which arranges all the spilled data on disk based on their arrival order. Clearly this storage pattern is simple, since we just need to append every newly spilled data at the end on disk. However, the disadvantage of this storage pattern is that the spilled data for multiple elements is mixed together. For instance, suppose both $title$ elements located on path $/auction/title$ and $bidder$ elements on path $/auction/bidder$ are flushed to disk. In this case, both the title elements and author elements will be recognized from the input stream and put together in their arrival order. Using this storage pattern, we would have to distinguish between these two elements again in the supplementary query to produce correct results. In addition, since the spilled data is sorted based on their arrival order, the I/O costs of reading disk data is proportional to the position of the spilled data. The elements which are spilled later would take longer time to read back into the memory.

In the future, to avoid such disk reading overhead and identification overhead, we can build an element indexing storage pattern. When the data is spilled into the disk, it is indexed based on the element name and its position in the input stream. For instance, we could index disk-resident elements using a vector (DocID, StartPos, EndPos, Level). A reference points to its physical location on the disk. In this case, we can locate the element quickly based on the index. The cost of recognizing each element is thus a constant value. In addition, the spilled data which belongs to the same document is placed on the same disk page to save the disk reading costs.

25.5 Load Spilling for XQuery with Window Functions

For data streams, since incoming data is infinite, storing the entire stream is obviously impossible. For many applications, data from the recent past is more likely to be relevant and interesting than older data. Continuous queries have been extended with sliding window constraints for relational streams [5] to purge stale data. A window constraint can be either time-based or count-based. A time-based window constraint indicates that only data that arrives within the last window time-frame is useful and need to be stored. A count-based window constraint indicates that only the most recent certain number of tuples need to be stored.

For XML streams, [14] first proposes to extend XQuery with a window function. In [14], a FORSEQ clause is proposed to represent windows using XQuery. Considering XML stream as an infinite sequence of items, the FORSEQ clause iterates over an input stream and binds the variable to a sub-sequence (aka window) of the input sequence in each iteration. An example window function (FORSEQ clause) is as follows:

```
DECLARE variable $seq as (string)**
FOR $a in $seq sliding window
  START at $x WHEN $seq[$x]/@a eq S
  END at $y WHEN $seq[$y]/@a eq E
  RETURN $w
```

The boundaries of a window are defined by START and END clauses. START and END clauses involve a WHEN clause which specifies a predicate. Intuitively, the WHEN condition of a START clause specifies when a window should start. The WHEN condition of a END clause specifies when an open window should be closed. The window function above generates subsequences of items from the input stream. It goes through input stream item by item. If the attribute a of an item is equal to “S”, a new window is opened.

If the attribute a of an item is equal to “E”, the open window is closed. Figure 25.3 shows an example fragment of input sequence and the windows generated by the above window function. Observe that three windows (subsequences) are generated. They are $\{b, c, d\}$, $\{c, d\}$ and $\{e, f\}$.

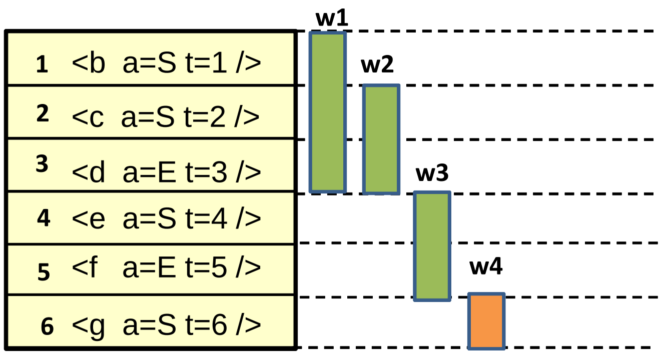


Figure 25.3: An Example Fragment of Input Sequence and Generated Windows

Let us go through the data fragment in Figure 25.3 and examine the generation of windows. For data shown in Figure 25.3, we iterate the input sequence. For the first item “b”, since its attribute a is equal to “S”, a new window $w1$ is opened. Similarly, for item “c”, since its attribute a is equal to “S”, window $w2$ is opened. For item “d”, since its attribute a is equal to “E”, windows $w1$ and $w2$ are closed. Here window $w1$ has subsequence $\{b, c, d\}$ and $w2$ has subsequence $\{c, d\}$. For item “e”, window $w3$ is opened. Window $w3$ is closed when item “f” is encountered. Note that window $w4$ is an open window since item “g” is the last item in the input sequence. In this case, such open window does not generate a subsequence.

Load spilling applied to XML streams with window functions brings new challenges. Note that spilling different items may lead to varying output of windows. Figure 25.4 and Figure 25.5 show the effect on output windows when spilling item “b” and “d” respectively. When spilling item “b”, since it is the start item of window $w1$, we cannot identify the start of window $w1$ in this case. Window $w1$ hence is not produced in the runtime output. When spilling item “d”, since its attribute a is equal to “E”, we cannot detect the end of windows $w1$ and $w2$. In this case, the output of these two windows is affected. Observe that both the spilling of “b” and “d” element cause the loss of some windows. This is because “b” as well as “d” element affect the predicate evaluation of START or END clause. Therefore, we need to measure how an input item contributes to the output of each window. In addition, algorithms must be designed to choose items to spill so to maximize the number of output windows.

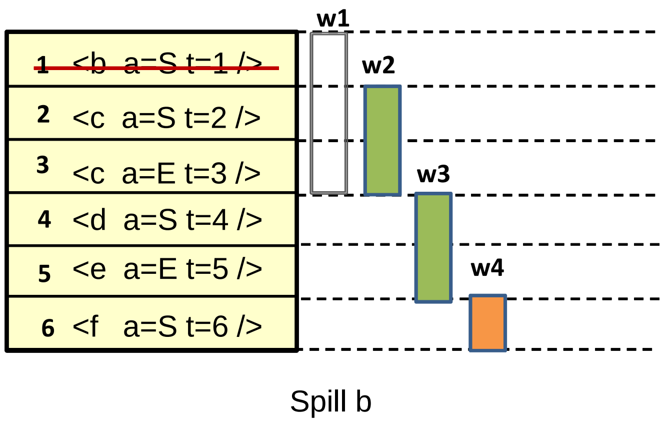


Figure 25.4: Spilling Effect on Windows When Spilling b

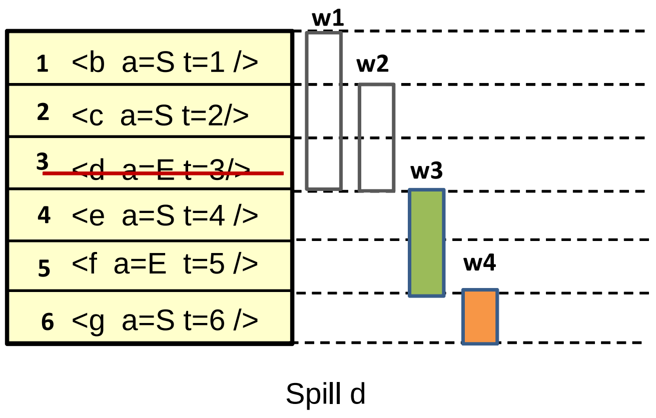


Figure 25.5: Spilling Effect on Windows When Spilling d

Bibliography

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, August 2003.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, and et. al. The design of the borealis stream processing engine. In *Proceedings of CIDR*, 2005.

[3] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient xml query pattern matching. In *IEEE International Conference on Data Engineering (ICDE)*, page 141, Feb 2002.

[4] M. Altinel and M. Franklin. Efficient Filtering of XML Documents for Selective Dissemination. In *Proceeding of VLDB*, pages 53–64, 2000.

[5] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[6] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *Proceedings of the ACM SIGMOD conference*, pages 261–272. ACM Press, 2000.

[7] A. M. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *ACM SIGMOD International Conference on Management of Data*, pages 419–430, 2004.

[8] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *vldb*, 13:333353, 2003.

[9] B. Babcock, S. Babu, R. Motwani, and J. Widom. Models and issues in data streams. In *PODS*, pages 1–16, June 2002.

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

[11] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *IEEE International Conference on Data Engineering (ICDE)*, pages 350–361, 2004.

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

[13] M. A. Bornea, V. Vassalos, Y. Kotidis, and A. Deligiannakis. Double index nested-loop reactive join for result rate optimization. In *IEEE International Conference on Data Engineering (ICDE)*, pages 481–492, 2009.

[14] I. Botan, P. M. Fischer, D. Florescu, D. Kossmann, T. Kraska, and R. Tamosevicius. Extending xquery with window functions. In *International Conference on Very Large Data Bases (VLDB)*, pages 75–86, 2007.

[15] C. Koch, S. Scherzinger, N. Scheweikardt and B. Stegmaier. FluxQuery: An Optimizing XQuery Processor for Streaming XML Data. In *International Conference on Very Large Data Bases (VLDB)*, pages 228–239, 2004.

[16] M. J. Carey, M. Blevins, and P. Takácsi-Nagy. Integration, web services style. *IEEE Data Eng. Bull.*, 25(4):17–21, 2002.

[17] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: a scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 379–390. ACM Press, 2000.

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

[19] M.-S. Chen, M.-L. Lo, P. S. Yu, and H. C. Young. Using segmented right-deep trees for the execution of pipelined hash joins. In *Proceedings of VLDB*, pages 15–26, 1992.

[20] M. E. Crovella, M. S. Taqqu, and A. Bestavros. Heavy-tailed probability distributions in the world wide web. In *In A Practical Guide To Heavy Tails, chapter 1*, pages 3–26. Chapman Hall, 1998.

[21] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *ACM SIGMOD International Conference on Management of Data*, pages 40–51, 2003.

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

[23] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer. Progressive merge join: A generic and non-blocking sort-based join algorithm. In *International Conference on Very Large Data Bases (VLDB)*, pages 299–310, 2002.

[24] F. Farag and M. A. Hammad. Adaptive execution of stream window joins in a limited memory environment. In *International Database Engineering and Applications Symposium*, pages 12–20, 2007.

[25] L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi. Query processing of streamed xml data. In *International Conference on Information and Knowledge Management (CIKM)*, pages 126 – 133, 2002.

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

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

[28] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Adaptive load shedding for windowed stream joins. In *International Conference on Information and Knowledge Management (CIKM)*, pages 171–178, 2005.

[29] A. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *ACM SIGMOD International Conference on Management of Data*, pages 419–430, 2003.

[30] H. Su, J. Jian and E. A. Rundensteiner. Raindrop: A Uniform and Layered Algebraic Framework for XQueries on XML Streams. In *CIKM*, pages 279–286, 2003.

[31] G. Häubl and V. Trifts. Consumer decision making in online shopping environments: The effects of interactive decision aids. *Marketing Science*, 19(1):4–21, 2000.

[32] Intel. XML Accelerator and XML Director. http://www.intel.com/support/netstructure/director, 2000.

[33] Z. Ives, A. Halevy, and D. Weld. An XML Query Engine for Network-Bound Data. *VLDB Journal*.

[34] Z. G. Ives, A. Y. Levy, D. S. Weld, D. Florescu, and M. Friedman. Adaptive query processing for internet applications. *IEEE Data Engineering Bulletin*, 23(2):19–26, 2000.

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

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

[37] W. Kießling, B. Hafenrichter, S. Fischer, and S. Holland. Preference XPATH- a query language for e-commerce. In *5th International Conference Wirtschaftsinformatic*, pages 427–440, 2001.

[38] W. Kießling and G. Köstler. Preference sql - design, implementation, experiences. In *International Conference on Very Large Data Bases (VLDB)*, pages 990–1001, 2002.

[39] R. Lawrence. Early hash join: a configurable algorithm for the efficient and early production of join results. In *International Conference on Very Large Data Bases (VLDB)*, pages 841–852, 2005.

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

[41] J. J. Levandoski, M. E. Khalefa, and M. F. Mokbel. Permjoin: An efficient algorithm for producing early results in multi-join query plans. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1433–1435, 2008.

[42] J. J. Levandoski, M. E. Khalefa, and M. F. Mokbel. Permjoin: An efficient algorithm for producing early results in multi-join query plans. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1433–1435, 2008.

[43] B. Liu, Y. Zhu, and E. Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *ACM SIGMOD International Conference on Management of Data*, pages 347–358. ACM Press, 2006.

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

[45] G. Luo, C. Tang, and P. S. Yu. Resource-adaptive real-time new event detection. In *SIGMOD Conference*, pages 497–508, 2007.

[46] M. F. Fernandez, D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *ICDE*, pages 14–23, 1998.

[47] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries on Heterogeneous Data Sources. In *Proceedings of the 27th VLDB Conference, Edinburgh, Scotland*, pages 241–250, 2001.

[48] M. Mokbel, M. Lu, and W. Aref. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *Proceedings of ICDE*, page 251, 2004.

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

[50] R. Motwani, J. Widom, A. Arasu, and et. al. Query processing, resource management, and approximation in a data stream management system. In *CIDR*, pages 245–256, 2003.

[51] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the Web. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Santa Barbara, CA*, pages 437–448, May 2001.

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

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

[54] J. Shanmugasundaram, K. Tufte, D. J. DeWitt, J. F. Naughton, and D. Maier. Architecting a network query engine for producing partial results. In *WebDB*, pages 17–22, 2000.

[55] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh based content routing using xml. In *SOSP*, pages 160–173, 2001.

[56] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *International Conference on Very Large Data Bases (VLDB)*, pages 324–335, 2004.

[57] H. Su, E. A. Rundensteiner, and M. Mani. Automaton in or out: Run-time plan optimization for xml stream processing. In *The Second International Workshop on Scalable Stream Processing Systems (SSPS'08)*, pages 17–22, 2008.

[58] Y. Tao, M. L. Liu, D. Papadias, M. Hadjieleftheriou, and N. Mamoulis. Rpj: producing fast join results on streams through rate-based optimization. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 371–382, New York, NY, USA, 2005. ACM.

[59] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *International Conference on Very Large Data Bases (VLDB)*, pages 309–320, 2003.

[60] N. Tatbul, U. etintemel, and S. B. Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *International Conference on Very Large Data Bases (VLDB)*, pages 159–170, 2007.

[61] N. Tatbul and S. B. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *International Conference on Very Large Data Bases (VLDB)*, pages 799–810, 2006.

[62] W. H. Tok, S. Bressan, and M.-L. Lee. A stratified approach to progressive approximate joins. In *EDBT '08: Proceedings of the 11th international conference on Extending database technology*, pages 582–593, New York, NY, USA, 2008. ACM.

[63] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.

[64] S. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *ACM SIGMOD*, pages 37–48, 2002.

[65] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, May 2003.

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

[67] M. Wei, E. A. Rundensteiner, and M. Mani. Utility-driven load shedding for xml stream processing. In *World Wide Web Conference*, pages 855–864, 2008.

[68] M. Wei, E. A. Rundensteiner, and M. Mani. Achieving high output utility under limited resources through structure-based spilling in xml streams. Technical report, Worcester Polytechnic Institute, 2009.

[69] M. Wei, E. A. Rundensteiner, and M. Mani. Achieving high output utility under limited resources through structure-based spilling in xml streams. In *International Conference on Very Large Data Bases (VLDB)*, pages 1267–1278, 2010.

[70] Y. Wu, J. M. Patel and H. V. Jagadish. Structural Join Order Selection for XML Query Optimization. In *IEEE International Conference on Data Engineering (ICDE)*, pages 443–454, 2003.

[71] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 30(2):425–436, June 2001.