NOTE

This online version of the thesis may have different page formatting and pagination from the paper copy held in the Swinburne Library.
A Query System for XML Data Streams and its Buffer Reduction based on Semantics

by

Chi Yang

B.E. (SDU, China)

A thesis submitted to
Faculty of Information and Communication Technologies
Swinburne University of Technology
for the degree of
MASTER BY RESEARCH

August, 2007
Abstract

The XML data sources are of marvellous diversity. Available XML data ranges from small Web pages to ever-growing applications, such as biological data, astronomical data, commercial data, and even to rapidly changing and possibly unbounded streams which are often encountered in Web data integration and online publish-subscribe systems. The ubiquity of XML data stream animates the importance to develop XML data stream query systems theoretically and practically. Currently, queries on XML data stream is still an active research area. The last years witnessed the work and contribution from both practitioners and theoreticians towards effective query evaluation against XML data streams. Due to the unique features of XML data stream model, several aspects of data management need reconsideration, such as buffer management, query optimization and block operation processing.

With respect to current methods for query evaluation over XML data stream, adoption of certain types of buffering techniques is inevitable. Under lots of circumstances, the buffer size may increase exponentially, which can cause memory bottleneck. Some theoretic research proving a concurrency lower bound to describe this type of bottleneck has been given. Some optimization techniques have been proposed to solve the problem. However, the research on semantic query optimization (SQO) for XML data streams is still at its early stage. In particular, the application of semantic information to optimize the buffering usage during the query evaluation leaves lots of room for researchers to maneuver.

The work reported in this thesis focuses on the study of query processing for XML data streams and effective buffer management by exploring semantic information. The first contribution is a SAX-based XML stream query evaluation system, which explores query optimization opportunities based on
semantics to reduce the unnecessary buffer scale to a level less than the theoretic lower bounds. The second contribution is that we get some effective semantic rules according to our criterion for rule exploration. Algorithms before and after the application of the semantic query optimization rules are presented and compared. The architecture of our system which deploys semantic optimization technique according to predefined stream optimization rules is shown. The third contribution is the further discussion on the application of SQO rules for block operators and aggregation functions. Experiments are conducted to demonstrate the system performance gains after the deployment of SQO techniques. The experiment results show that the algorithms deploying semantic rules individually and collectively all significantly outperform the lower bound algorithm that does not consider semantic information.
Acknowledgements

During the last two years, many people have contributed directly or indirectly to the development of this research topic. Their contribution is critical to the accomplishment of my thesis. I would like to appreciate their help and support, and give my gratitude to them.

First of all, I have to appreciate the continuous trust, instruction and support from my supervisor, Associate Professor Chengfei Liu whose firm mind, intensive knowledge and flexible methodology for carrying out research impressed me deeply. Without the input and inspire from Associate Professor Chengfei Liu, it is impossible to finish such a difficult work by myself as a fresh man into the research field.

The gratitude also should be given to my associate supervisors, Professor Yun Yang, Dr. Jinjun Chen for their help and support for completing this thesis and related work. All of their advices have a constructive guide to my research work.

The thanks should also be given to Professor Jeffrey Xu Yu and Dr. Junhu Wang for their constructive suggestion about my work.

Furthermore, I am grateful to other research students Jianxin Li, Xiaohui Zhao and Rui Zhou with whom the author often made discussion about research oriented topics and the thesis writing. Sometimes, the discussion result benefits my research work.

Here I also want to express my thanks to my dear friend Xin Liu, whose meticulous accompany and suggestion for both my life and research encouraged and inspired me so much during last one year for finishing the difficult work that composes this thesis.

Finally, the thanks have to be given to my parents who supported me continuously to my research and endured the long time separation, which gave me huge confidence and determination to fulfil the tough work during the
evolution of this thesis. Then let me say thanks to all the researchers and friends who spent every night in lab accompanying me during last two years.
Declaration

This thesis contains no material which has been accepted for the award to the candidate of any other degree or diploma, except where due reference is made in the text of the thesis. To the best of the candidate’s knowledge, this thesis contains no material previously published or written by another person except where due reference is made in the text of the thesis.

Chi Yang

August 2007
List of Figure

Figure 1.1   Comparison between data streams and database systems.............. 2
Figure 2.1   Concurrency of $D \text{ w.r.t. } Q=a[p]/b[c]/e$................................. 15
Figure 2.2   DFA for XPath query evaluation.................................................. 22
Figure 2.3   NFA for XPath query evaluation................................................... 22
Figure 2.4   Indexed query structure from NFA.............................................. 23
Figure 2.5   SPEX for query evaluation step by step...................................... 23
Figure 3.1   XML document example scenario.............................................. 27
Figure 3.2   XML document example scenario.............................................. 28
Figure 3.3   Standards and tools for XML document processing..................... 29
Figure 3.4   XML document fragment for ‘book’ element............................... 30
Figure 3.5   APIs in JAXP.............................................................................. 32
Figure 3.6   Example document for DOM representation............................... 33
Figure 3.7   DOM tree representation of the example document..................... 33
Figure 3.8   Functioning model of SAX.......................................................... 37
Figure 3.9   Comparing StAX to other JAXP APIs.......................................... 40
Figure 3.10  DTD tree examples................................................................. 43
Figure 3.11  Sequence control from DTD....................................................... 44
Figure 3.12  Choice control from DTD............................................................ 45
Figure 3.13  Optional and repeatable element.............................................. 46
Figure 4.1   Architecture of SwinXSS............................................................. 51
Figure 4.2   Forward XPath............................................................................ 52
Figure 4.3   Grammar of Simplified Forward XPath....................................... 53
Figure 4.4   Vector representation of query tree............................................ 55
Figure 4.5   XML stream processor implementing SAX interface.................... 57
Figure 4.6   ContentHandler interface of SAX.............................................. 58
Figure 4.7   Basic buffer management in SwinXSS........................................ 60
Figure 5.1   Relationship between CONCUR($D,Q$) and document layers...... 65
Figure 5.2   Influence after deployment of Predicate After Rule...................... 68
Figure 5.3   Influence after deployment of Predicate Ahead Rule.................... 69
Figure 5.4   Influence after deployment of Maximum Cardinality Rule.......... 69
Figure 5.5   Influence after deployment of Co-exist Rule.............................. 70
Figure 5.6   Influence of semantic rules to buffer management....................... 71
Figure 5.7   Buffer management optimization.............................................. 76
Figure 6.1   Block operation query scenario.................................................. 79
Figure 6.2   Block operation over the XML stream....................................... 80
Figure 6.3   XPath carrying block operation predicates.................................. 80
Figure 6.4   The influence from document towards update operation.............. 81
Figure 7.1   XMark document structure fragment........................................ 85
Figure 7.2   XMark document structure fragment........................................ 86
Figure 7.3   Test queries................................................................................ 87
List of Figures

Figure 7.4   Maximum buffer scale for 1GB and 2GB XML dataset.............. 89
Figure 7.5   Using the semantic information for Concur lines breaking........ 89
Figure 7.6   Response time for 1GB and 2GB XML dataset............................ 90
Chapter 1: Introduction

XML data streams combine the flexible structure of XML data and specific features of stream data together, form a new research frontier. In this chapter, we first give a brief introduction for both XML and data streams, then list the important issues for processing queries over XML data streams. Finally, the organization and contributions of this thesis are presented.

1.1 XML (Extensible Mark-up Language)

XML (Extensible Mark-up Language) is a well known standard for electronic data interchange on the Web nowadays. Soon, XML was proved to be powerful with respect to document publishing and data interchange format after it was first released in 1998 by the World Wide Web Consortium (W3C). XML, as a subset of SGML, has a superficial resemblance to HTML (Hyper Text Mark-up Language), the established language of the Web. But information held in XML format is self-describing which means the XML format information can be extracted, manipulated and formatted to the requirements of any target audience, publishing medium or XML-enabled software application. So compared with HTML, XML is of interest which offering great flexibility to manage and customize the documents. The above advantages make XML popular as a data format almost everywhere and of course you can easily find XML described information in your everyday life.

Due to the ultimate application of XML data, there are lots of XML related techniques such as Namespaces, Schema, DTD, XSLT, CSS, XPointer, XPath,
Chapter 1. Introduction

DOM and SAX. Compared with relational databases, XML has more complex and flexible structures.

1.2 Data Stream

Another important concept is data streams which come from real applications. In those applications the data is modelled best not as persistent relations but rather as transient data streams.

<table>
<thead>
<tr>
<th></th>
<th>Data streams</th>
<th>Databases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>transient</td>
<td>permanent</td>
</tr>
<tr>
<td>Queries</td>
<td>continuous</td>
<td>transient</td>
</tr>
<tr>
<td>Updates</td>
<td>append-only</td>
<td>arbitrary</td>
</tr>
<tr>
<td>Answers</td>
<td>approximate</td>
<td>accurate</td>
</tr>
<tr>
<td>Data manipulating</td>
<td>single-passed filtering</td>
<td>arbitrary</td>
</tr>
<tr>
<td>Query plans</td>
<td>adaptive</td>
<td>static</td>
</tr>
<tr>
<td>Indexing</td>
<td>of queries</td>
<td>of data</td>
</tr>
</tbody>
</table>

**Figure 1.1** Comparison between data streams and database systems

Generally speaking, a data stream system can be described as multiple transient relations or structures, each of which consists of infinite tuple sequence or defined data items. The data update method of data streams is append-only. With the respect to query processing, queries on data streams are usually continuous with approximate answers. In addition, query evaluation takes the form of one pass, and the query plan is adaptive. By contrast, a traditional relational database can be modeled as persistent relations for holding finite sets of tuples. The stored data can be modified and updated arbitrarily and the queries are transient with exact query answers. The query evaluation can be arbitrarily and the query plan is fixed. The data model determines that great differences exist unavoidably between designs of DSMS and DBMS. Figure 1.1 shows that via a comparison. To support these new
features from data streams, several aspects of database management need to be reconsidered by database researchers.

Examples of such applications include financial applications, network monitoring, security, telecommunications data management, web applications, manufacturing, sensor networks, XML based online publishing and others. In the data stream model, individual data items may be relational tuples, e.g., network measurements, call records, web page visits, sensor readings or XML elements marked with start tags and end tags, and so on. The main application domains of data streams include:

- **Sensor-based monitoring systems, e.g., for traffic or atmospheric conditions**
  New techniques for monitoring data streams are developed to locate objects, like cars on highways or luggage in airports, which are equipped with position emitters (sensors). For example, in meteorology, streams of scalar XML values representing atmospheric conditions are gathered by sensors and used in monitoring systems that enable, e.g., early recognitions of tornados. The Sensor Web project [NJ04] at NASA develops instruments for monitoring and exploring environments.

- **Usage monitoring systems**
  Streams conveying transactional data are gathered over networks from credit card usages and phone calls for detecting usage patterns indicating possible frauds [CFPRS00].

- **Publish-subscribe systems, e.g., for press, media, or financial news**
  The world becomes increasingly information-driven and the natural need to find the desired information is associated to finding the needle in a haystack. To partially fulfil these needs, publish-subscribe systems [AF00] are used to selectively disseminate existing information gathered from various heterogeneous sources (e.g., newspapers) called publishers.
Chapter 1. Introduction

• **XML packet routing**
XML routers perform content-based routing of individual XML packets to other routers or clients based upon profiles (queries) that describe the information needs [GSC01]. Industry trend towards development of XML messaging systems for business applications has already intrigued start-up development of XML routing systems in companies, e.g., Firano Software, Sarvega, Forum, Elitesecureweb, Knowhow, Xbridgesoft, XMLblaster [GS03].

• **Video filtering based on XML content descriptions**
The new generation video standards, e.g., MPEG-7 [MTR02, VTR01], provide elaborate XML content descriptions that contain information ranging from “size” to the “current speaker in scene”. Such metadata is to be transmitted as an XML stream separated yet related with the real video stream. The content-based video filtering and routing is needed, e.g., for jumping directly to or skipping certain scenes. First several prototypes of MPEG-7 based systems, e.g., [RHK03], require efficient filtering techniques for fast and continuous XML streams of highly structured metadata.

• **Analysis of scientific data**
The European Southern Observatory (ESO) [MWF04] is confronted with the problem of processing weekly terabytes of astronomical data, as gathered by its Very Large Telescope (VLT). Such raw (pixel-based) data is usually accompanied by its content description (metadata) wrapped in XML. The characteristics of this kind data are of typical data streams. Its arrival rate and size make a standard approach for storing, indexing, and processing rather difficult.

1.3 Queries over XML Data Streams

Query problems are basic and critical for any database system. It is the same to data stream processing. Queries over continuous XML data streams have much
in common with queries in querying XML documents. Some techniques in querying XML documents can be geared for addressing the query evaluation problem in XML data streams. In turn, the techniques from querying the XML data streams enrich the query processing for XML documents. However, there are two important distinctions between the query processing in XML documents and XML data streams.

The first distinction is between one-time queries and continuous queries. One-time queries (a class that includes traditional DBMS queries) are queries that are evaluated once over a point-in-time snapshot of the data set, with the answer returned to the user. Continuous queries, on the other hand, are evaluated continuously as data streams continue to arrive. Continuous queries are the more interesting class of data stream queries, and it is to them that we will devote most of our attention. The answer to a continuous query is produced over time, always reflecting the stream data seen so far. Continuous query answers may be stored and updated as new data arrives, or they may be produced as data streams themselves.

The second distinction is between predefined queries and ad hoc queries. A predefined query is one that is supplied to the data stream management system before any relevant data has arrived. Predefined queries are generally continuous queries, although scheduled one-time queries can also be predefined. Ad hoc queries, on the other hand, are issued online after the data streams have already begun. Ad hoc queries can be either one-time queries or continuous queries. Ad hoc queries complicate the design of a data stream query system, both because they are not known in advance for the purposes of query optimization, identification of common sub-expressions across queries, etc., and more importantly because the correct answer to an ad hoc query may require referencing data elements that have already arrived on the data streams (and potentially have already been discarded). All the above existing problems have identical reflection both in the XML data streams and tuple-based data streams. The reason lies in that they are the problems from stream modelling regardless what the data format is.

To effective query an XML data stream we identify the following issues:
Chapter 1. Introduction

- Due to the nature of streaming XML data, it queries a large amount of memory to evaluate a query precisely. How to minimise the memory utilisation is a challenging task.
- Semantic query optimization is a popular method for query optimization over XML documents. Recently, it gains a new application toward XML data streams. The main reason is that under many circumstances, the uncertainty of the streaming XML data in future can be made clear by semantic information from DTD/Schema. Consequently, the query processing can be optimized.
- Block operation is another interesting research problem because it requires a set of data from XML streams. How to get that data set to finish the computation defined by the block operators is still an open topic.

1.4 Thesis Contributions and Organization

The work presented in this thesis is motivated by the above three query problems. Briefly, it attempts to build up an XML stream query system for solving those problems. The rest of this thesis is organized in seven chapters. We list them here and highlight our contributions.

Chapter 2 gives a literature review of previous projects and works within the data stream research area. Consequently, the main existing issues and research topics related to XML and data streams will be introduced.

Chapter 3 recalls widely accepted models and syntaxes of semi-structured data, among which the tree model and the XML syntax are used in this work. For query languages over XML data streams, we will have a brief review of XPath. A sub-section of XPath named Forward XPath will be introduced. The Java based XML document processing techniques will also be given. Among them we will focus on SAX in JAXP which acts an important role in most of XML stream processing engines for the sake of its push API feature. Then XML Schema and DTD (document type definition) will be briefly reviewed, several scenarios of semantics within them will be described. The current
Chapter 1. Introduction

Theoretic lower bounds for XML streams processing is critical and will be introduced in this chapter with the current state of art of XML SQO techniques.

Chapter 4 focuses on our XML data stream querying system architecture and corresponding implementation. Specifically, we explain the different components and techniques of our system, including Forward XPath, query definition, a forward XPath parser with semantics, the query tree with semantics in SwinXSS, the SAX based events processing engine, buffer management mechanism, the basic algorithm with Concurrency Lower Bound [YFJ04, YFJ05], the algorithm strategies and entry points for possible optimizations. After that, an application scenario will show the basic algorithm for the original query plan of our query system.

Chapter 5 proposes several semantic rules for optimising the buffer usage during the query evaluation process. In detail, according to the theoretic lower bounds, we give some theoretic criteria for finding semantic rules. All the rules we give can be derived from our criterion. The algorithms to implement those rules for forming the adaptive query plan are given. Based on the algorithm in chapter 4 and this chapter, an example run will be used to demonstrate the significant buffer reduction after the deployment of our semantic optimization rules. How our semantic rules contribute to break the limitation of theoretic lower bound can be seen here.

Chapter 6 explores an interesting topic “block operators” within the XML data stream query evaluation framework. Under most of database systems or query systems, block operation is an inevitable topic. In this chapter, we will show our effort to borrow the idea from chapter 4 and 5, to curb the possible scale of buffer resulting from block operators as small as possible with the help of semantic information. The related semantic rule will also be given.

Chapter 7 shows our system performance and efficiency. We test our system performance using the generated XMark standard documents. The query selectivity given is based on the fragments of generated XMark documents. The experimental data will be analysed. Through the empirical study, it is
proved that our SwinXSS can effectively outperform the theoretic lower bounds as the result that we expect.

Finally, in Chapter 8, we conclude our work. Based on that, we outline some interesting issues for future discussion. In Appendix of the thesis, the code of SwinXSS engine is given to fuel the interest of a critical eye with some proofs which are skipped from the main body of this work.
Chapter 2: Literature Review

Recent years witnesses the tremendous efforts towards processing XML data streams both theoretically and empirically. Although there still exist many open research topics, the research about XML data streams and its related technologies has generated fruitful results. In the rest of this chapter, we take a brief retrospect of them.

2.1 Review of Data Stream Projects

A lot of data stream processing systems have been introduced with different application purposes and specialties. Following is the overview of several past and current projects related to data stream management and data stream processing. We will revisit some of these projects which can be treated as important references when we discuss the issues that we are facing in constructing our XML data stream querying and filtering system.

Continuous queries were used in the Tapestry system [TGNO92] for content-based filtering over an append-only database of email and bulletin board messages. A restricted subset of SQL was used as the query language in order to provide guarantees about efficient evaluation and append-only query results. The Alert system [SPAM91] provides a mechanism for implementing event-condition-action style triggers in a conventional SQL database, by using continuous queries defined over special append-only active tables.

The Tribeca stream database manager [SM96] provides restricted querying capability over network packet streams. The Tangram stream query processing
system [PMC89, PSV92] uses stream processing techniques to analyze large quantities of stored data.

The *OpenCQ* [LPT99] and *NiagaraCQ* [CDTW00] systems support continuous queries for monitoring persistent data sets spread over a wide-area network, e.g., web sites over the Internet. OpenCQ uses a query processing algorithm based on incremental view maintenance, while NiagaraCQ addresses scalability in number of queries by proposing techniques for grouping continuous queries for efficient evaluation. Within the NiagaraCQ project, Shanmugasundaram et al. [STDNM00] discuss the problem of supporting blocking operators in query plans over data streams, and Viglas and Naughton [VN02] propose *rate-based optimization* for queries over data streams, a new optimization methodology that is based on stream-arrival and data-processing rates.

The *Chronicle data model* [JMS95] introduced append-only ordered sequences of tuples (*chronicles*), a form of data streams. They defined a restricted view definition language and algebra (*chronicle algebra*) that operates over chronicles together with traditional relations. The focus of the work was to ensure that views defined in chronicle algebra could be maintained incrementally without storing any of the chronicles. An algebra and a declarative query language for querying ordered relations (*sequences*) was proposed by Seshadri, Livny, and Ramakrishnan [SLR94, SLR95, SLR96]. In many applications, continuous queries need to refer to the sequencing aspect of streams, particularly in the form of sliding windows over streams. Related work in this category also includes work on temporal [SA85] and time-series databases [FRM94], where the ordering of tuples is implied by time can be used in querying, indexing, and query optimization.

The body of work on materialized views relates to continuous queries, since materialized views are effectively queries that need to be reevaluated or incrementally updated whenever the base data changes. Of particular importance is work on *self-maintenance* [BCL89, GJM96, QGMW96] ensuring that enough data has been saved to maintain a view even when the base data is unavailable and the related problem of *data expiration* [MLY98] determining
Chapter 2. Literature Review

when certain base data can be discarded without compromising the ability to maintain a view. Nevertheless, several differences exist between materialized views and continuous queries in the data stream context: continuous queries may stream rather than store their results, they may deal with append-only input data, they may provide approximate rather than exact answers, and their processing strategy may adapt as characteristics of the data streams change.

The Telegraph project [AH00, HF00, MF02, MHSR02] shares some target applications and basic technical ideas with a DSMS. Telegraph uses an adaptive query engine (based on the Eddy concept [AH00]) to process queries efficiently in volatile and unpredictable environments (e.g., autonomous data sources over the Internet, or sensor networks). Madden and Franklin [MF02] focus on query execution strategies over data streams generated by sensors, and Madden et al. [MHSR02] discusses adaptive processing techniques for multiple continuous queries. The Tukwila system [IFFLW99] also supports adaptive query processing, in order to perform dynamic data integration over autonomous data sources.

The Aurora project [CCCCLS02] is building a new data processing system targeted exclusively towards stream monitoring applications. The core of the Aurora system consists of a large network of triggers. Each trigger is a data-flow graph with each node being one among seven built-in operators (or boxes in Aurora’s terminology). For each stream monitoring application using the Aurora system, an application administrator creates and adds one or more triggers into Aurora’s trigger network. Aurora performs both compile-time optimization (e.g., reordering operators, shared state for common subexpressions) and run-time optimization of the trigger network. As part of run-time optimization, Aurora detects resource overload and performs load shedding based on application-specific measures of quality of service.

The XFilter content-based filtering system [AF00] performs efficient filtering of XML documents based on user profiles expressed as continuous queries in the XPath language [XPath1.0]. The YFilter [DFFT02] shows its scalability and efficiency using a FSA-based execution model when querying the XML data streams. Xyleme [NACP01] is a similar content-based filtering...
system that enables very high throughput with a restricted query language. Raindrop system has been developed to tackle the challenges of stream processing XQuery queries based on both automata models and algebraic models. The SPEX system [OMFB02, ODF03, FT03, SD03, OFB04-1, OFB04-2, BCDFOS05], processes a considerably large XPath fragment containing all axes with polynomial complexities. However, most of the current XML stream systems are based on FSM theory: given a set of queries, the first class of systems report on matched queries against XML documents conveyed in the stream [AF00, IHW02, CFGR02, CGORS02, DAFZF03, DFFT02, GMOS03, GS03], whereas others return the matched stream fragments [LMP02, OKB03, PC03-2, PC03-1, BCGRFJ02, FT03, SD03, BCGRFJ03, KS03, OFB04-1, OFB04-2, KSS04-2, KSS04-1, BCDFOS05, PC05]. Considering the automata-based characteristics, for processing large sets of queries, most of them employ various techniques for detecting commonalities among queries. They differ mainly in the complexities of the employed query evaluation algorithm, which vary from linear [CFGR02, BCGRFJ02, OFB04-1, OFB04-2] to exponential [AF00, CGORS02, PC03-1] in the size of the queries, and in the degree of supporting XPath or XQuery fragments for specifying queries, which varies from simple XPath paths with child and descendant axes [AF00, IHW02, CFGR02, CGORS02, PC03-1] to XQuery queries with child and descendant axes and result construction [LMP02, KS03, KSS04-2].

2.2 Unbounded Memory Requirements

Since data streams are potentially unbounded in size, the amount of storage required to compute an exact answer to a data stream query may also grow without bound. While external memory algorithms [VJ99] for handling data sets larger than main memory have been studied, such algorithms are not well suited to data stream applications since they do not support continuous queries and are typically too slow for real-time response. The continuous data stream model is most applicable to problems where timely query responses are important and there are large volumes of data that are being continually
produced at a high rate over time. New data is constantly arriving even as the old data is being processed; the amount of computation time per data element must be low, or else the latency of the computation will be too high and the algorithm will not be able to keep pace with the data stream. For this reason, we are interested in algorithms that are able to confine themselves to main memory without accessing disk. Arasu et al. [ABBMW02] took some initial steps towards distinguishing between queries that can be answered exactly using a given bounded amount of memory and queries that must be approximated unless disk accesses are allowed. They consider a limited class of queries and, for that class, provide a complete characterization of the queries that require a potentially unbounded amount of memory (proportional to the size of the input data streams) to answer. Their result shows that without knowing the size of the input data streams, it is impossible to place a limit on the memory requirements for most common queries involving joins, unless the domains of the attributes involved in the query are restricted (either based on known characteristics of the data or through the imposition of query predicates). The basic intuition is that without domain restrictions an unbounded number of attribute values must be remembered, because they might turn out to join with tuples that arrive in the future. Extending these results to full generality remains an open research problem. With respect to XML data streams processing, some work toward the theoretic lower bounds can be found. However, due to the possible unbound features of the stream itself, the complexity can increase to an unpredictable level.

2.2.1 Theoretic Lower Bounds for XML Stream Processing

As mentioned in the previous section, for XML data stream processing, the theoretic research results are fruitful. Among them, there are very instructive and important lower and upper bounds which have been proved, according to different real application environment or streaming data model constraints.

As an important standard for the system consumption, the Lower bounds problems are critical. It is also a measurement for us to conceive our algorithm
and conclude our semantic rules. We will introduce them in the rest of this section for the sake of their importance.

All known algorithms \[PC03-1, PC03-2, GS03, LMP02\] for evaluating XPath and XQuery queries over XML streams suffer from excessive memory usage on certain queries and documents. The bulk of memory used is dedicated to two tasks: (1) storage of large transition tables; and (2) buffering of document fragments. The former emanates from the standard methodology of evaluating queries by simulating Finite-state automata. The latter is a result of the limitations of the data stream model. Finite-state automata or transducers are the most natural mechanisms for evaluating XQuery/XPath queries. However, algorithms that explicitly compute the states of these automata and the corresponding transition tables incur memory costs that are exponential in the size of the query in the worst-case. The high costs are a result of the blowup in the transformation of non-deterministic automata into deterministic ones. In those papers seeing \[YFJ05\] some work investigated the space complexity of XPath evaluation on streams as a function of the query size, and showed that the exponential dependence is avoidable. An optimal algorithm whose memory depends only linearly on the query size (for some types of queries, the dependence is even logarithmic) has been exhibited. The other major source of memory consumption has also been studied: buffers of (representations of) document fragments. Algorithms that support advanced features of the XPath language, such as predicates, full-fledged evaluation (as opposed to only filtering), or closure axes, face the need to store fragments of the document stream during the evaluation. The buffering seems necessary, because in many cases at the time the algorithm encounters certain XML elements in the stream, it does not have enough information to conclude whether these elements should be part of the output or not (the decision depends on unresolved predicates, whose final value is to be determined by subsequent elements in the stream). Indeed, all the advanced evaluation algorithms maintain some form of buffers (e.g., the stack of the XPush Machine \[GS03\], the BPDT buffers of the XSQ system \[PC03-1\], the predicate buffers of TurboXPath \[JFB04\], and the buffer trees of the FluX query engine
Chapter 2. Literature Review

[KSS04-1, KSS04-2]). It has been noted anecdotally that for certain queries and documents, buffering seems unavoidable. However, up to date, formal and theoretical study that quantifiers the amount of buffering needed to support advanced features of XPath has been proved.

With respect to the advanced features of XPath, two major classes of XPath evaluation problems that necessitate buffering. Space lower bounds that quantify the amount of buffering required in terms of some document properties such as none-recursive document, star free etc.

Bar-Yosseff et al. [YFJ04, YFJ05] investigated the upper two types of space complexity of XPath evaluation on streams and proved that for any algorithm $A$ that evaluates a star free XPath query $Q$ on an XML streaming document $D$, the minimum bits of space that $A$ needs to use. These two types of space complexity are so call theoretic lower bounds for the query processing over XML data streams. These two classes of evaluation problems are the following:

2.2.2 Full-fledged Evaluation of Queries with Predicates

There are two typical modes of query evaluation: full-fledged evaluation (i.e., outputting all the document elements selected by the query) and filtering (i.e., outputting a bit indicating whether the query selects any node from the document or not). It is proved that full-fledged evaluation of queries with predicates requires substantially more space than just filtering documents using such queries. This concurrency lower bound is based on full-fledged evaluation. The lower bound is stated in terms of a property of documents called “concurrency” denoted as $\Omega(CONCUR(D,Q))$. This lower bound is defined on the concept of a concurrency.

![Figure 2.1](image)

**Figure 2.1** Concurrency of $D$ w.r.t. $Q=a[p]/b[c]/e$
As shown in Figure 2.1, document $D$ is represented as a stream of 16 events called time steps. The concurrency of the document $D$ with respect to query $Q$ at step $t \in [1, m]$ is the number of content-distinct nodes in $D$ that are alive at step $t$. As shown in Figure 1, let $Q=a[p]/b[c]/e$. At step 14, two $e$ elements are alive. The first is at step 3 because whether it will be selected depends on whether its $a$ grandparent will have a $p$ child. The second is at step 13, because whether it will be selected depends on whether its $b$ parent will have a $c$ child and its $a$ grandparent will have a $p$ child. So the concurrency at step 14 is 2.

The document concurrency of $D$ w.r.t. $Q$, denoted as $\text{CONCUR}(D,Q)$, is the maximum concurrency over all steps $t \in [1, m]$. For example, $\text{CONCUR}(D,Q)$ in Figure 1 is 2. The concurrency lower bound is suitable for single variable predicate queries. For queries with a multi-variable predicate, the dominance lower bound is defined in [YFJ05]. It is simple to verify that if $Q$ is a non-predicate query, $\text{CONCUR}(D,Q)$ is 1. However, for queries with single predicate, it is easy to construct documents with arbitrarily large concurrency.

### 2.2.3 Multi-variate Comparison Predicates

A predicate that consists of a comparison of two nodes (e.g., $a = b$ or $a > b$) is said to be a multi-variate comparison predicate. On the other hand, univariate comparison predicates are ones that compare a node with a constant (e.g., $a = 5$ or $b > 4$). It has been proved that evaluation (whether full-fledged or filtering) of queries that consist of multi-variate comparison predicates may require substantially more space than evaluation of queries that have only univariate comparison predicates.

The existential semantics of XPath implies that a predicate of the form $/c[R(a,b)]$, where $R$ is any comparison operator (e.g., $=, >$), is satisfied if and only if the document has a ‘c’ node with at least one 'a' child with a value $x$ and one ‘b’ child with a value $y$, so that $R(x; y) = true$. Thus, if all the ‘a’ children of the ‘c’ node precede its ‘b’ children, the evaluation algorithm may need to buffer the (distinct) values of the ‘a’ children, until reaching the first ‘b’ child. It is proved that such a buffering is indeed necessary when $R$ is an equality
operator (i.e., =,!=). It is not needed for inequality operators (i.e., <,<=,>,>=), because for them it suffice to buffer just the maximum or minimum value of the ‘a’ children.

The lower bound is in fact stated w.r.t. to any relational operator $R$, not just comparisons. The bound is given in terms of a graph-theoretic property of relations, which is called the “dominance cardinality”. So this type of lower bound names dominance lower bound.

**Theorem:** For the query $Q = /c[R(a,b)]$, for every integer $k$, and for every filtering algorithm $A$ for $Q$ on XML streams, there exists a document $D$ of candidate cardinality $k$ w.r.t. $u$ on which $A$ uses $\Omega(\log \text{DOM}_k(R))$ bits of space.

The theorem of above dominance lower bound clarifies the minimum space consumption clearly for any query with several predicates belonging to the same query node.

Note that for the equality operators (=,!=), the above lower bound is $\Omega(\log(\frac{|U|}{k})) = \Omega(\log(|U|))$ for sufficiently small $k$'s. That means that if $R$ is an equality operator, evaluation of $Q$ on documents that have a node with $k$ children that match $u_a$ (or $u_b$) would require buffering the distinct data values of these children. On the other hand, when $R$ is an inequality operator (<,<=,>,>=), evaluation of $Q$ requires only $\Omega(\log|U|)$ bits of space, which is what is needed to buffer the maximum or minimum data value of the $k$ children that match $u_a$ (or $u_b$).

The detailed definitions and proof of two space lower bounds can be found in [YJF04, YJF05].

### 2.3 Query Techniques

As described in the Section 1.3 and Section 2.2, when we are limited to a bounded amount of memory it is important to select the most suitable and optimal strategy of buffer management for data stream queries. However,
Chapter 2. Literature Review

under the circumstances where the exact query answers can not be got, high-quality approximate answers are often acceptable. Algorithms for query problems defined over data streams has been a fruitful research area recently. This work has led to some general techniques for data reduction and synopsis construction, including: sketches [AMS96, FM83], random sampling [PGA00, RPGA99, CMN99], histograms [IP99], and wavelets [CCMN00, VW99]. Based on these summarization techniques, we have seen some work on approximate query answering. For example, recent work [DGGR02, GKS01] develops histogram-based techniques to provide approximate answers for correlated aggregate queries over data streams, and Gilbert et al. [GKMS01] present a general approach for building relatively small space summaries over data streams to provide approximate answers for many classes of aggregate queries. However, research problems abound in the area of approximate query answering, with or without streams. Even the basic notion of approximations remains to be investigated in detail for queries involving more than simple aggregation.

The research towards querying algorithms has been fairly active in the area of data streams, typically motivated by problems in databases and networking. The model of computation underlying the algorithmic work can be stated as follows:

The main complexity measure is the space used by the algorithm, although the time required to process each stream element is also relevant. And the space consumption can be categorized into two types, one is total space usage, the other is maximum space usage.

Though work in [HRR98] defined a similar model but also allowed the algorithm to make multiple passes over the stream data, making the number of passes itself a complexity measure. Most of models restrict the attention to algorithms which are allowed only one pass. In the next several subsections, we will list several popular approaches for query processing.
2.3.1 Sliding Windows

One technique for producing an approximate answer to a data stream query is to evaluate the query not over the entire past history of the data streams, but rather only over sliding windows of recent data from the streams. For example, only data from the last week could be considered in producing query answers, with data older than one week being discarded. The work for Sliding Window can be found here in [BDM02, DGIM02]. In querying XML data stream, the adoption of Sliding windows is somehow difficult. The reason is that tree structure relationship of the XML document can be lost due to sliding windows. The work toward sliding window algorithms can also be found in [DGIM02, BDM02, MM02, GK02, GKS01].

2.3.2 Batch Processing

Batch processing is a type of query technique to produce approximate answers to give up on processing every data element as it arrives. The representative research of this type can be found in [UF00].

2.3.3 Sampling

Based on possibility and statics theory, sampling technique catches up with the speed of incoming data stream at the expense of accuracy to some extend. Unfortunately, for many situations (including most queries involving joins [CM99, CMN99, HHW97]), sampling-based approaches cannot give reliable approximation guarantees. The theoretic work for algorithms such as [MR95, AGPR99, AGP00, CDN01, VJ85, CMN99] can also be found for random samples.

2.3.4 Synopsis Data Structures

With respect to types of data stream queries where no exact data structure with the desired properties exists, one can often design an approximate data structure that maintains a small synopsis or sketch of the data rather than an exact representation, and therefore is able to keep computation per data element to a minimum. To construct a synopsis data structure, some algorithms
can be selected, such as [AMS96, CMN00, HNSS95, DGGR02, GKS01, FKS99, IP00] for sketching technique, [FSMMU98] for histograms, [MVW98, VW98, VW99, CGRS00] for wavelets.

2.3.5 Queries Referencing Past Data
In the data stream model of computation, once a data element has been streamed by, it cannot be revisited. This limitation means that ad hoc queries that are issued after some data has already been discarded may be impossible to answer accurately. One simple solution to this problem is to stipulate that ad hoc queries are only allowed to reference future data: they are evaluated as though the data streams began at the point when the query was issued, and any past stream elements are ignored (for the purposes of that query). While this solution may not appear very satisfying, it may turn out to be perfectly acceptable for many applications. The research on ad hoc queries has been seen in [AH00, CN97].

2.3.6 Blocking Operation
A blocking query operator is a query operator that is unable to produce the first tuple of its output until it has seen its entire input. Sorting is an example of a blocking operator, as are aggregation operators such as SUM, COUNT, MIN, MAX, and AVG. If one thinks about evaluating continuous stream queries using a traditional tree of query operators, where data streams enter at the leaves and final query answers are produced at the root, then the incorporation of blocking operators into the query tree poses problems. Since continuous data streams may be infinite, a blocking operator that has a data stream as one of its inputs will never see its entire input, and therefore it will never be able to produce any output. Clearly, blocking operators are not very suitable to the data stream computation model, but aggregate queries are extremely common, and sorted data is easier to work with and can often be processed more efficiently than unsorted data. Doing away with blocking operators altogether would be problematic, but dealing with them effectively is one of the more
Chapter 2. Literature Review

challenging aspects of data stream computation. Some of approaches have been offered to handling block operators such as work in [RRH99, TMSF02].

The fruitful research results can be seen in relational database community about block operators. Some work has dedicated to categorize the operators into different types, and for different types operators the processing strategies are different. The above concept can not be directly used to process XML data streams’ block operators, because the tree like structure and SAX events model of XML data streams make the original category from traditional database not suitable to process correspondent block operators. So if the correct categorization can be done and offered, the processing efficiency and capability of block operators against XML data streams will of course be dramatically enhanced.

Under XML stream environment, the processing of block operators is a unique challenge for the sake of the XML document structure. Generally, within XML data streams, a block operator requires the computations on one or a series of sub-trees. How to extract data belonging to the sub-trees and how to choose more optimized strategies are also interesting topic for querying XML data stream. To our knowledge, the work towards this field is pretty limited due to the dramatic increase of complexity when buffers for block operation are unavoidable.

2.4 Querying and Filtering XML Streams

For query processing over XML data streams, lots of finite state machines (FSM) or automata based filtering algorithms [AF00, AH00, BCDFOS05, DFFT02] have been introduced. FSMs are a natural and effective way to represent and process path expressions. Elements of a path expression are mapped to states. A transition from an active state is fired when an element is found in the document that matches that transition. If an accepting state is reached, then the document is said to satisfy the query. For example, XFilter [AF00], has focused on the efficient evaluation of path expressions over streaming XML data, using indexed Finite State Machines (FSM) to allow many structure-based queries to be processed simultaneously. However,
Chapter 2. Literature Review

XFilter, makes no attempt to eliminate redundant processing for similar queries. Figure 2.2 shows the DFA based techniques for processing multi-queries. The query trees are finally transformed into some definite states for processing.

![DFA for XPath query evaluation](image)

**Figure 2.2** DFA for XPath query evaluation

YFilter [DFFT02, DAFZF03] combines multiple queries into a single Nondeterministic Finite Automaton (NFA). The use of a combined NFA allows a dramatic reduction in the number of states needed to represent the set of user queries and greatly improves filtering performance by sharing execution work. YFilter also extends this NFA model to efficiently handle predicates within path expressions. Figure 2.3 shows that, a set of queries (a) have been transformed into a NFA (b) for query evaluation.

![NFA for XPath query evaluation](image)

**Figure 2.3** NFA for XPath query evaluation
Chapter 2. Literature Review

The NFA then will be indexed as Figure 2.4, which is used to match the elements from the incoming stream.

**Figure 2.4** Indexed query structure from NFA

*SPEX* [BCDFOS05] system adopts a progressively match all the elements with query one by one, to form the output query result. The SPEX system will evaluate the predicated during the matching of the incoming elements. As shown in figure 2.5, the query node (b) can be visited several times and a data node (a) at most can be visited once by a query node.

\[
\langle a_1 \rangle \langle a_2 \rangle \langle c_1 \rangle \langle c_1 \rangle \langle a_2 \rangle \langle c_2 \rangle \langle c_2 \rangle \langle d_1 \rangle \langle d_1 \rangle \langle a_1 \rangle / \text{desc:a[child:d]}/\text{desc:c}
\]

**Figure 2.5** SPEX for query evaluation step by step

2.5 Semantic Query Optimization over XML Data Streams

Semantic Query Optimization (SQO) is comparatively a recent approach for the transformation of given query into equivalent alternative query using matching rules in order to select an optimal query based on the costs of
executing alternative queries. To understand the SQO we have to introduce two important concepts of **semantically equivalent** and **semantic transformation**.

1. Two queries are **semantically equivalent** if they return the same answer for any database state satisfying a given set of integrity constraints.
2. A **semantic transformation** transforms a given query into a semantically equivalent one.

Semantic query optimization is the process of determining the set of semantic transformations that results in a semantically equivalent query with a lower execution cost. The discussion of optimization is indispensable in terms of any query algorithm and its corresponding query plan execution. Lots of optimization techniques have been offered, and semantic query optimization (SQO) is one of them.

XML stream-specific optimization techniques have been well developed. In this demo, it aims to highlight the schema-based optimization (SQO) on one abstraction level. Schema knowledge is used to rewrite a query into a more efficient one. Most current literature on SQOs in XML focuses on techniques that are either (1) general regardless of persistent or streaming XML sources or (2) specific to persistent XML source. For example, query tree minimization is a general technique. It eliminates a pattern from the query if the pattern is known to always exist. Since the pruned query involves less computation than the original one, it is more efficient to evaluate regardless of the nature of data sources. For another example, the query rewriting using state extents technique requires indices the data. Applications on persistent XML can usually afford the preprocessing of building indices while this is often not the case for XML stream applications due to the on-the-fly arriving nature of their data. Therefore this technique is more suitable for persistent XML.

Being rich in semantic elements and expressions is an important feature of XML language itself, which means SQO can be used to optimize the query on XML document, consequently, on XML data streams. The work of [SRM04,
Chapter 2. Literature Review

SRM05] focuses on SQOs specific to XML streams instead of XML documents. To our knowledge, their work has proposed a comprehensive solution for XML stream specific SQO techniques. It handles only limited query (i.e., Boolean XPath or XQuery match) with one type of constraint. In contrast, firstly, they handle more complex query type, i.e., a subset of XQuery. Secondly, their system support most commonly used constraints in XML Schema. However, the application of SQO on XML data streams is at its initial stage. Some progress can be seen in the work [SRM04, SRM05], but that is far from completion and soundness.

Actually, SQO for XML streams is not an isolated topic, its high related to most of the query evaluation problems in XML streams, such as lower bounds, block operators and referencing past data problems. In later Chapter 5 and Chapter 6 we will show how SQO are blended into query processing to optimize the solving of those issues.

2.6 Summary

In this chapter, we review the previous projects related to XML data streams. Some theoretic results of memory consumption and space lower bounds are also reviewed. Then we introduced the popular methods and algorithms for answering queries, including Sliding Window, Random Samples, Histograms, block operation etc. After that, the filtering approaches for querying XML streams developed are introduced and compared. Finally, we introduce the current research and application stage of SQO over the processing of XML data streams.
Chapter 3: Preliminary

Before the description of our work, we first introduce some basic knowledge and techniques related to our work.

3.1 XML

XML adopts a semi-structure data [ABS00, CSY01, CFGR02, AF00, CB01-1, CB01-2] model, differentiates from other structures. To better understanding the rest of thesis, we give it a concise retrospect in this section.

3.1.1 Grammars for XML Data

Although XML data has an implicit structure, given by the labels stored within the tags, it is often useful to specify further structural and content constraints for XML documents. Such constraints can be specified within grammars (often called schemata) that define languages of well-formed XML documents. The XML documents generated by a grammar G are valid with respect to G, or simply G-valid. The advantages offered by the existence of grammars for XML documents stem mainly from the data structure and content awareness that can be used, e.g., by basic services like storage and querying for improving efficiency. There are several formalisms for specifying XML grammars. Among them, Document Type Definition (DTDs) [BPMM98], XML-Schema [FW01], and Relax NG [CM01] are the most popular ones. All these formalisms are special subclasses of regular tree grammars [LMM00], thus the
theory of regular tree grammars and of tree automata, to which tree grammars are related, can be fruitfully used also for studying the properties of the practical aforementioned grammar languages. Directly derived from the membership problem for tree automata, [LMM00] develops also validation tools for XML documents.

3.1.2 Example of XML

We introduce here two real-life scenarios of semistructured data for expressing in XML marking-up.

Since the arrival of the XML syntax for semistrucured data, the common practice in processing data across networks is to deal locally with robust database systems that handle relational data and to wrap it in XML, when it comes to exchange it. Our first scenario considers a natural relational structure expressed using semistructured data. This scenario models a journal archive as a node-labeled tree, where each journal is represented as a node with label journal, and each of its properties, like title, editor, authors, and price, are represented as children nodes with corresponding labels. Figure 3.1 shows a possible journal entry and its XML serialization.

```
<journal>
  <title>db</title>
  <editor>dan</editor>
  <authors>
    <name>ana</name>
    <name>bob</name>
  </authors>
  <price>7</price>
</journal>
```

Figure 3.1 XML document example scenario

Semistructured data is also used in practical cases to express tree structures with recursive definition. The second scenario considers a real-life case of semistructured data expressing the genealogical (or family) tree of important
historical persons, like pharaohs, kings, or emperors. Such tree data were described since ancient times, and eventually used to decide on the successors at thrones. This scenario models the genealogical tree of someone’s folk (ancestors, descendants, brothers and sisters, nephews and nieces) as a node-labeled tree, where that person is represented as the root node, and each other person is represented as a node with label either man or woman and has a child text node consisting in its name, e.g., in the case of John this would be ‘John’.

![XML document example scenario](image)

**Figure 3.2** XML document example scenario

The children of a person are represented also as children nodes of the node corresponding to that person, and the order between these nodes reflects the ascending order of the age of the corresponding children. An interesting instance of this scenario is the family tree of the kings of France. An excerpt from its third dynasty, i.e., the Valois dynasty (1328-1589), is simplistically...
modelled in Figure 3.2 starting with the king John II, the Good, and ending shortly before the ascension to the throne by Louis XII in 1498.\footnote{This family tree is in fact a graph: Louis XI is the son of Marie of Anjou and Charles VII, the Dauphin, and Charles of Valois is the son of Valentina Visconti and Louis of Valois.}

3.2 XPath

The popular standards for traversing the XML documents are recommended by W3C including XPath, XQuery, etc. We will focus on the XPath standard in the rest of this section.

Under most of the XML real applications, it is vital to navigate through XML documents, including querying of XML documents to find and receive required material, the creation of hypertext links to objects that do not have unique identifiers, merging of documents and document fragments, and the formatting of document components for presentation. To meet the above application need, comes the XPath standard.

XPath is a language for finding information in an XML document. XPath is used to navigate through elements and attributes in an XML document.

XPath is a major element in the W3C's XSLT standard- and XQuery and XPointer are both built on XPath expressions. Actually, it has even been incorporated into Schema and XInclude directly or indirectly. The relationship between those XML standards and XPath can be seen in figure 3.3.

![Figure 3.3 Standards and tools for XML document processing](image-url)
Chapter 3. Preliminary

Naturally, an understanding of XPath is fundamental to a lot of advanced XML usage. The meaning of an element can depend on its contextual location. For example, a *Title* element that is embedded within a *Book* element has a different meaning to one that is embedded within a *Name* element. The format will certainly differ, and a query that is used to extract a list of book titles should not include entries as ‘Mr’, ‘Dr’ or ‘Miss’. So every element in an XML document has a specific and unique contextual location. The hierarchical and sequential structures in an XML document can be used as stepping stone, and any element in the document can be identified by the steps it would take to reach it, either from the root element, or from some other fixed starting location. For example, the last name of the author of a book may be held in a specific instance of a *Name* element. It could be obtained by stepping through the ‘book’, ‘front’, ‘author’, ‘name’, and ‘last’ elements. This would select the name ‘Smith’, but not the name ‘Jones’ from the document fragment as shown in Figure 3.4.

```xml
<book>
  <front>
    <author>
      <name>
        <first> John </first>
        <last> Smith </last>
      </name>
    </author>
  </front>
  …
  <!---NOT SELECTED--->
  <chapter> <name> <init> F </init> <last> Jones </last> </name>
</book>
```

*Figure 3.4* XML document fragment for ‘book’ element
Chapter 3. Preliminary

To meet the requirement of unique query result locating, XPath is an answer for its step pointing syntax (See in W3C Website) and expression similar to UNIX, Ms-DOS file system directories. Review this scenario example, to select the name ‘Smith’, we can use XPath expression `/book/front/author/name/first AND last` where the symbol ‘/’ means a step towards a certain direction.

XPath is easy to use and quite flexible, but that doesn’t keep XPath from rich expressions. XPath expressions are used in different ways for different purposes. Some expressions are location paths, used to identify and extract, link to or re-use targeted information in the document. Some location path can take the role of a pattern, and be used simply to confirm or deny that an element of interest is in a specific location in the document (the XSLT standard makes use of this concept).

XPath can express both single and multiple selections. Most of XPath expressions select multiple elements in a document. However it is possible to create expressions that are guaranteed to only select a specific element instance, or simply to determine whether an instance actually exists in the document.

XPath expressions allow the adoption of predicates and filters to qualify any step within the XPath. Location paths are quite indiscriminate. For example, the path ‘book/chapter/pare’ selects all of the paragraphs in all of the chapters. But there is often a need to target a more selective set of elements, and possibly a single element instance. A predicate filter is used to qualify any step in the path. The list of matches at each step is reduced by asking questions about the nodes in this list. Square brackets are used to hold the predicates. For example, carrying predicate expression, ‘book/chapter[contenttest()=1]/pare’ will only select all the paras under chapter 1.

3.3 JAXP

The Java API for XML Processing, or JAXP, is one of the Java XML programming APIs. It provides the capability of validating and parsing XML documents. The three parsing interfaces are:

- the Document Object Model parsing interface or DOM interface
Chapter 3. Preliminary

- the Simple API for XML parsing interface or SAX interface
- StAX the Streaming API for XML. JSR #173.

Figure 3.5 APIs in JAXP

The relationship between different interfaces of JAXP can be seen in figure 3.5, it is clear that within JAXP framework, SAX often acts as a base for other APIs including DOM and StAX, etc. Each API has its own advantages and disadvantages according to different XML data processing requirement. In 2.3.1 to 2.3.3, we list and compare the functions and features of APIs in JAXP to reason our choice, SAX for streaming XML data items parsing. In addition to the parsing interfaces, the API provides an XSLT interface to provide data and structural transformations on an XML document. JAXP was developed under the Java Community Process as JSR 5 (JAXP 1.0) and JSR 63 (JAXP 1.1 and 1.2). J2SE 1.4 is the first version of Java that comes with an implementation of JAXP. As of 2006, the current version of JAXP is version 1.2 and JAXP 1.3 is being developed under JSR 206.

There are two major types of XML (or SGML) APIs within JAXP framework, one is tree based APIs, the other is event based APIs.

3.3.1. Tree-based APIs

These map an XML document into an internal tree structure, then allow an application to navigate that tree. The Document Object Model (DOM) [WR00] working group at the World-Wide Web Consortium (W3C) maintains a recommended tree-based API for XML and HTML documents, and there are
many such APIs from other sources. Tree-based APIs are useful for a wide range of applications, but they normally put a great strain on system resources, especially if the document is large. Furthermore, many applications need to build their own strongly typed data structures rather than using a generic tree corresponding to an XML document. It is inefficient to build a tree of parse nodes, only to map it onto a new data structure and then discard the original.

We will demonstrate the usage and working model of tree based APIs through the explanation of its typical representative, DOM.

What is the Document Object Model?

The DOM is a programming API for documents. It closely resembles the structure of the documents it models. For instance, consider this document fragment in figure 3.6, taken from an XML document:

```
<TABLE>
  <TBODY>
    <TR>
      <TD>Shady Grove</TD>
      <TD>Aeolian</TD>
    </TR>
    <TR>
      <TD>Over the River, Charlie</TD>
      <TD>Dorian</TD>
    </TR>
  </TBODY>
</TABLE>
```

Figure 3.6 Example document for DOM representation

The DOM API represents this table like this in figure 3.7:

```
<TABLE>
  <TBODY>
    <TR>
      <TD>Shady Grove</TD>
      <TD>Aeolian</TD>
    </TR>
    <TR>
      <TD>Over the River, Charlie</TD>
      <TD>Dorian</TD>
    </TR>
  </TBODY>
</TABLE>
```

Figure 3.7 DOM tree representation of the example document
However, DOM will maintain a document tree within the memory for operation and manipulation. Though DOM API can offer the clients the flexibility to accesss the data items within XML document randomly, the memory tree structure representation requires to see the end of whole document. So DOM is not suitable for possibly endless XML data streams.

### 3.3.2 Event-based APIs

An event-based API, on the other hand, reports parsing events (such as the start and end of elements) directly to the application through callbacks, and does not usually build an internal tree. The application implements handlers to deal with the different events, much like handling events in a graphical user interface. SAX is the best known example of such an API. An event-based API provides a simpler, lower-level access to an XML document: you can parse documents much larger than your available system memory, and you can construct your own data structures using your callback event handlers.

Consider the following task:

*Locate the record element containing the word "Ottawa".*

If your XML document were 20MB large (or even just 2MB), it would be very inefficient to construct and traverse an in-memory parse tree just to locate this one piece of contextual information; an event-based interface would allow you to find it in a single pass using very little memory.

To understand how an event-based API can work, consider the following sample document:

```xml
<?xml version="1.0"?>
<doc>
    <para>Hello, world!</para>
</doc>
```

An event-based interface will break the structure of this document down into a series of linear events, such as these:
An application handles these events just as it would handle events from a graphical user interface: there is no need to cache the entire document in memory or secondary storage.

Finally, it is important to remember that it is possible to construct a parse tree using an event-based API, and it is possible to use an event-based API to traverse an in-memory tree.

**SAX Interface**

SAX is the *Simple API for XML*, originally a Java-only API. SAX was the first widely adopted API for XML in Java, and is a “de facto” standard. The current version is SAX 2.0.1, and there are versions for several programming language environments other than Java.

SAX has recently switched over to the SourceForge project infrastructure. The intent is to continue the open development and maintainence process for SAX (no NDAs required) while making it easier to track open SAX issues outside of the high-volume xml-dev list. Project resources include archived mailing lists and a download area. See the Project Page link for full information about project facilities which are being used, as well as news announcements. David Megginson, who runs an XML consulting company, has resumed maintaining SAX after a period of excellent work by David Brownell.

In order to understand the SAX API clearly, two important concepts should be explained first. One is Call-backs, the other is Event handlers.
Chapter 3. Preliminary

Call-backs: The application instantiates a parser object, supplied by the parser developer (in this case the open-source Apache Foundation (www.apache.org) parser called Xerces), and instructs it to parse a specified XML document or data stream. As the parser processes the XML data, it detects significant items, such as an element start-tag, or a comment. But the parser also needs to send all this information back to the main application. The SAX API is mainly concerned with the means by which the parser is able to return information to the application, using a “call-back” mechanism. The parser must be given access to methods in objects created by the main application, so that it can pass-on information as it reads it from the XML document.

Event handlers: Depending on the need, the application creates the one or more objects that contain methods that the parser must call when appropriate events occur. The application may only wish to be informed of document markup: the element start-tags, end-tags and text content (as well as any whitespace), the comments and processing instructions, and the document start and end. But it may also wish to be informed of calls to external unparsed entities, and declarations of notations. Or it may also wish to be informed of errors. When errors occur, it may need to be able to determine the location of these errors. Finally, it may wish to be informed of all calls to external entities, and be given the opportunity to intercept and redirect these calls.

The application passes references to these “handler” objects to the parser, so that it can call the methods in them when appropriate. Following figure 3.8 shows the SAX functioning model. In this model, the application or Main() function will create a Parser to read the XML document and divide it into different defined units. Then the corresponding event handlers will be called by the parser to react to the specific SAX events from document.
So that the parser can accept the handler objects created by the application, they must belong to a class that defines one or more of the SAX interfaces (located at ‘org.xml.sax.*’). The parser can then be certain that the necessary methods are present, and call them when appropriate events occur.

The typical SAX interfaces are:

- `Parser` (implemented by the parser itself) (deprecated in SAX 2.0)
- `DocumentHandler` (deprecated in SAX 2.0)
- `AttributeList` (deprecated in SAX 2.0)
- `ErrorHandler`
- `EntityResolver`
- `Locator`
- `DTDHandler`

There are some important activities involved in parsing, navigating and processing documents that the SAX standard does not cover. All the softwares implementing SAX standard have to provide a parser for XML document or data stream. Apache provides classes for parsing an XML file or data streams, including one called “SAXParser”. This class will create that parser.

SAX is unique among XML APIs in that it models the parser rather than the document. In particular the parser is represented as an instance of the `XMLReader` interface. The specific class that implements this interface varies from parser to parser. Most of the time you only access it through the common methods of the `XMLReader` interface. A parser reads a document from
beginning to end. As it does so it encounters start-tags, end-tags, text, comments, processing instructions, and more. In SAX, the parser tells the client application what it sees as it sees it by invoking methods in a DocumentHandler object. Within Main() function in figure 3.8, there is an object implementing the XMLReader interface which actually is an XML parser to decompose XML format data sequence. During decomposing, XMLReader object will call the methods in DocumentHandler to make reaction to different predefined SAX events, such as #PCDATA, start and end tags etc.

DocumentHandler is a critical interface which the client application implements to receive notification of document content. The client application instantiates a client-specific instance of the DocumentHandler interface and registers it with the XMLReader that’s going to parse the document. As the reader reads the document, it calls back to the methods in the registered DocumentHandler object. The DocumentHandler is important for realizing our query system, the indispensable methods which should be implemented within DocumentHandler interface including `startDocument()`, `endDocument()`, `characters()`, `startElement()`, `endElement()`, `endPrefixMapping()`, `ignorableWhitespace()`, `setDocumentLocator()`, `skippedEntity()`, `startPrefixMapping()`.

However, due to the simplicity and event based parsing, SAX requires less support and decomposes the XML document into units one by one, which means SAX is very basic and makes it very suitable for the stream environment (XML data stream model modifies the stream as continuous data items or units arriving one by one which is quiet coincident to the SAX API’s working pattern). SAX as a standard for many years, it is proved to be reliable and efficient; at the same time, the techniques pertaining to it are mature and well established. So we choose the SAX based event parsing model to build up our query platform.

### 3.3.3 StAX Standard

The Streaming API for XML (StAX), is a streaming Java-based, event-driven, pull-parsing API for reading and writing XML documents. StAX enables you
to create bidirectional XML parsers that are fast, relatively easy to program, and have a light memory footprint.

StAX provides is the latest API in the JAXP family, and provides an alternative to SAX, DOM, TrAX, and DOM for developers looking to do high-performance stream filtering, processing, and modification, particularly with low memory and limited extensibility requirements.

Based on the previous knowledge of SAX and DOM, we give comparison towards them to explain the StAX.

Comparing StAX to Other JAXP APIs
As an API in the JAXP family, StAX can be compared, among other APIs, to SAX, TrAX, and JDOM. Of the latter two, StAX is not as powerful or flexible as TrAX or JDOM, but neither does it require as much memory or processor load to be useful, and StAX can, in many cases, outperform the DOM-based APIs. The same arguments outlined above, weighing the cost/benefits of the DOM model versus the streaming model, apply here.

With this in mind, the closest comparisons between can be made between StAX and SAX, and it is here that StAX offers features that are beneficial in many cases; some of these include:

- StAX-enabled clients are generally easier to code than SAX clients.
  While it can be argued that SAX parsers are marginally easier to write, StAX parser code can be smaller and the code necessary for the client to interact with the parser simpler.
- StAX is a bidirectional API, meaning that it can both read and write XML documents. SAX is read only, so another API is needed if you want to write XML documents.
- SAX is a push API, whereas StAX is pull. The trade-offs between push and pull APIs outlined above apply here.

Figure 3.9 synopsizes the comparative features of StAX, SAX, DOM, and TrAX
Chapter 3. Preliminary

<table>
<thead>
<tr>
<th>Feature</th>
<th>StAX</th>
<th>SAX</th>
<th>DOM</th>
<th>TrAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Type</td>
<td>Pull, streaming</td>
<td>Push, streaming</td>
<td>In memory tree</td>
<td>XSLT Rule</td>
</tr>
<tr>
<td>Ease of Use</td>
<td>High</td>
<td>Medium</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>XPath Capability</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CPU and Memory Efficiency</td>
<td>Good</td>
<td>Good</td>
<td>Varies</td>
<td>Varies</td>
</tr>
<tr>
<td>Forward Only</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Read XML</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Write XML</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Create, Read, Update, Delete</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Figure 3.9** Comparing StAX to other JAXP APIs

**Pull Parsing Versus Push Parsing**

Streaming *pull parsing* refers to a programming model in which a client application calls methods on an XML parsing library when it needs to interact with an XML infoset—that is, the client only gets (pulls) XML data when it explicitly asks for it.

Streaming *push parsing* refers to a programming model in which an XML parser sends (pushes) XML data to the client as the parser encounters elements in an XML infoset—that is, the parser sends the data whether or not the client is ready to use it at that time.

Pull parsing provides several advantages over push parsing when working with XML streams:

- With pull parsing, the client controls the application thread, and can call methods on the parser when needed. By contrast, with push processing, the parser controls the application thread, and the client can only accept invocations from the parser.
- Pull parsing libraries can be much smaller and the client code to interact with those libraries much simpler than with push libraries, even for more complex documents.
- Pull clients can read multiple documents at one time with a single thread.
Chapter 3. Preliminary

- A StAX pull parser can filter XML documents such that elements unnecessary to the client can be ignored, and it can support XML views of non-XML data.

Though the StAX combines the advantages of both DOM and SAX APIs, but most of tools to implement it are based on SAX, which actually masks the SAX detailed processing and intra-event function calls to the programmer. So StAX is not so primitive or basic under the streaming environment. Additionally, it is a newly introduced API, some aspects of its performance still needs to be proved in real applications. Based on above consideration, we don’t adopt the StAX for our system setting up.

3.4 XML DTD and Schema

The semantic information used for XML data streams query optimization mainly comes from DTD and Schema information. Here we give a brief introduction of the subsections of them which are highly related to our work in Chapter 5 and Chapter 6.

3.4.1 XML DTD (document type definition)

Much of the XML standard is dedicated into the concept of document modelling. A DTD can be used to ensure that the documents confirm to predefined rules. In other words, it is possible to define in advance which elements can be used within a document, and where in the document these elements may be employed. This concept is similar to, but more powerful than, the stylesheet mechanisms that all modern word processors and desktop publishing packages provide.

Actually, there are SGML DTDs that describe HTML 2.0, HTML 3.2 and now HTML 4.0. These DTDs are not compatible with XML, primarily because they use some of the additional features of SGML. There is now also an XML DTD for the new XHTML standard.
XML DTDs use the same syntax as SGML DTDs, but have a more limited scope (Chapter 32 covers the differences in detail). Also note that XML Schema standard re-introduces many of the features omitted from XML DTDs.

**Models of XML DTD**

A document model specifies the names of the elements that may be used in a document. It also dictates which elements have element content, mixed content and text content, and which elements are empty. When elements can contain other elements, restrictions on the content may include the names of the elements allowed, and even the order in which they may occur. The names of attributes each element may hold are given and constraints may be placed on the possible value of an attribute.

The document modelling scheme provided by the standard is used to create a DTD. But DTD is not a single object; this is the collective name for a model built from definitions in a number of markup declarations.

DTDs are typically used to describe such document types as journals, training guides, technical manuals and reference books, as well as to help define other standards that utilize XML syntax.

For modelling the DTD, we can also use the tree structure. But compared with XML document, the tree is rarely identical. Unlike a document tree, the model tree contains a branch for all the options, and does not include repeating elements.

In the example below, the model tree specifies that a book contains a number of Chapter elements, with each Chapter containing either a number of Paragraph elements, or a single Sections element. A particular document tree, on the other hand, will have a branch for each Chapter in the book, and any one of these chapters would typically contain a number of paragraph elements. The tree for modelling the DTD in this example can be seen in figure 3.10.
Chapter 3. Preliminary

Figure 3.10 DTD tree examples

When to use a DTD

- Creation of a DTD is rarely a trivial exercise, but there are several important benefits to having one:
- Characteristics of a set of documents can be codified in a formal, unambiguous manner, for later of reference.
- Programmers can write extraction and manipulation filters without fear of their software ever having to process unexpected input.
- Stylesheets can be written with the same degree of confidence because a style rule can be defined for each element, and each context of significance, defined in the DTD.
- Using an XML sensitive word processor, authors and editors can be guided and constrained to produce conforming documents.

XML DTD can also specify the names of attributes, and assigns them to specific elements. It specifies whether an attribute is optional or required, and what kind of value it may hold. An attribute can hold a simple text phrase, a token, list of tokens, hypertext link anchor, or a reference to an entity or notation.

An attribute may also have a default value, to be applied only if no value is supplied by the document author. For example, a DTD may define Version and Date attributes for the Book element, and a Section Number attribute for the Section element. The Version attribute value may default to ‘1.0’.

XML DTD has its own markup and declarations. Each declaration has to confirm to the markup format, and is classified using one of the following keywords.
Chapter 3. Preliminary

- Element (tag definition)
- ATTLIST (attribute definitions)
- ENTITY (entity definition)
- NOTATION (data type notation definition)

Declarations are grouped together, and may be held within the internal subset of the document type declaration:

```xml
<!DOCTYPE MYBOOK [
<!--The MYBOOK DTD appears here-->
<!...........>
<!.....................>
]>
```

This approach makes the DTD part of the document it describes. This can be convenient. If the document is moved, the DTD automatically moves with it, as only a single data file or stream is involved.

We will give several element declarations which are highly related to our work here.

Sequence control

The sequence rule ‘(a,b,c)’ within DTD element declaration indicates that element ‘a’ is followed by element ‘b’, which in turn is followed by element ‘c’, as shown in figure 3.11. All three elements will be indispensable in the document.

Figure 3.11 Sequence control from DTD

Note that other markup, such as comments and processing instructions, may be inserted between these elements. Such markup is not part of the document.
Chapter 3. Preliminary

structure. In an article, for example, it may be important for the title to appear first, followed by the author’s name, then a summary.

...(title, author, summary)...

<article>
</article>

Choice control
The choice rule ‘(a | b | c)’ indicates a choice between the elements ‘a’, ‘b’ and ‘c’. Figure 3.12 describes this type of relationship.

![Choice control from DTD](image)

**Figure 3.12** Choice control from DTD

For example, an article in a magazine may be factual or fictional:

...(fact | fiction)...

<article>
  <fact>...</fact>
</article>
Quantity control
The DTD author can also dictate how often an element can appear at each location. If the element is required and may not repeat, no further information is required. All of the previous examples indicated a required presence (except where the ‘|’ connector specified a choice of element). It is also a simple matter to specify a fixed number of occurrences of an element. For example, if every article in a magazine had three authors, the following model would ensure that three names are present:

...author, author, author,...

But is also possible to make an element optional, to allow it to repeat any number of times, and even to make it both optional and repeatable. These occurrence rules are governed using quantity indicators (the symbols ‘?’ , ‘*’, and ‘+’).

Optional and repeatable element
If an element is optional, and also repeatable, the element name is followed by an asterisk, ‘*’. The ‘*’ may be seen as equivalent to the combination ‘?+’. For example, ‘(a, b*)’ indicates that element ‘b’ may occur any number of times, and may also be absent. Figure 3.13 shows this declaration.

Figure 3.13  Optional and repeatable element
Chapter 3. Preliminary

An article element may contain any number of Author elements, including none. To take another example, a chapter may have preliminary paragraphs, but may not always do so. Following XML fragment shows this phenomenon.

```xml
<chapter>
  <para>...</para>
  <para>...</para>
  <para>...</para>
  <section>...</section>
</chapter>
<chapter>
  <section>...</section>
</chapter>
```

That is the basic definition and functions of XML DTD. For detailed declarations of DTD, the W3C site can be used as the reference.

3.4.2 XML Schema

Because the expressive capability of DTD is limited, the schema information can be used sometimes as semantic information resource. We will touch the core of XML schema, the aspects indispensable with respect to our work will also be discussed here.

The term schema is used to describe all of the alternative modelling languages. In the IT field, this name has its roots in database technologies. It is used to describe the tables and fields in a database, including constraints on the values of particular fields, and the relationships between tables. More generally, the term simply means a representation of something using a diagram, a plan or an outline.

In May 2001, the W3C released the XML Schema standard. This standard had a very long gestation and this is not surprising, as the aim was to create a single modelling language that would please all interested parties.
In terms of features, XML Schema models are backward compatible with DTDs. This is very important, for the practical reason that it eases the transition from DTD modelling to XML Schema modelling. It is always possible to convert a DTD into an XML Schema model automatically. At the same time, XML Schema can offer the compliment to the XML DTD, the content includes functionality, namespace sensitivity support, data exchange applications. Finally, the syntax of XML Schema is self-description, which stands for using XML document to describe the XML Schema.

**Schema document structure**

Because an XML Schema model is an XML document (unlike DTD), it must be enclosed by a root element, in this case, the Schema element:

```xml
<schema...>
...
</schema>
```

The Schema element has a Version attribute, but this attribute is not used to identify the version of XML Schema in use; it is instead used by schema authors to identify the version of the schema, assuming that the schema undergoes periodic updates.

All schema elements have an Id (identifier) attribute. These attributes have no purpose within the schema language, and are included for convenience only. For example, it could be anticipated that hypertext links may be created from an XML document that documents the schema, with each link targeting the element in the schema that is currently under discussion.

Without the root element, there can first be any number and combination of inclusions, imports, re-definitions and annotations, followed by any number of combinations of simple and complex data type definitions. Following is an example of XML Schema for XML document structure and element content modelling.

```xml
<schema...>

<!--ANY NUMBER OF FOLLOWING-->
```
Chapter 3. Preliminary

For element definitions, XML Schema ‘Element’ is used to define an element. The capability of ‘Element’ can offer more constraints than what XML DTD can do. The specific power of element definitions include Simple content, Complex content, Empty elements, Child element content, Namespace, Occurrence options, Choices, Group occurrence options, Embedded groups, Mixed content etc. For detailed description of XML Schema application, it can be found in [W3C Website].

3.5 Summary

In this chapter we gave an introduction to all the indispensable knowledge and work for querying XML data streams including XML essential knowledge, XPath standard, JAXP tools, XML DTD/Schema. And with that we come to our work in Chapter 4.
Chapter 4: SwinXSS - A Platform for Querying XML Data Streams

Most of XML data stream query algorithms are based on automata theory as we introduced in Chapter 2. In order to avoid the disadvantages from the automaton based algorithms, we use a different method. That processes the matching directly according to the navigation sequence of the XML streams step by step. Based on this method, we build up a query system for XML data streams called Swinburne XML Stream System (SwinXSS). SwinXSS acts as a platform for our research work about semantic query optimization over XML data streams.

Figure 4.1 shows the SAX-based architecture of SwinXSS. Three different components of the system will be discussed in the following sub-sections according to their functions. One important input for SwinXSS is “XPath query expressions”. Forward XPath Parser is the component for processing user queries and transforming them into generated query trees. The other input of the parser is DTD/Schema, which will be discussed in Chapter 5 for semantic query optimization. Query Tree is a data structure representing user queries to instruct the work of the SAX-Based Processor. With the help of the Buffer Manager, The SAX-Based Processor matches all the elements from the incoming stream according to the query tree, and generates the output stream.
4.1 Simplified Forward XPath Language

Because of the “stream” characteristics, to support the full XPath for XML data stream is difficult. Similar to the work from [YJF05], we also simplify the XPath2.0 to get Forward XPath, formally a conjunction of Univariate XPath, Subsumption-free XPath and Symmetric XPath. It supports the forward axes only.

4.1.1 Symmetric XPath

- Symmetric XPath means all predicates are symmetric
- symmetric predicate: evaluation on any document node is independent of the order of node’s children

For example, article[year=2003] is a symmetric XPath expression because there is no order requirement for the node year. Whereas, article/year[last()] is not symmetric because of the ‘last()’ function.
Chapter 4. SwinXSS-A Platform for Query XML Data Streams

4.1.2 Univariate XPath

- all atomic predicates are univariate
- *atomic predicate*: maximal subexpression without boolean connectors
- *univariate predicate*: depends on the value of at most one node in the query tree

For example, `/author/article[(year>2003) and (count(author)>2)]` is a univariate XPath because both the atomic predicates connected by “and” include only one variable. Whereas `//article[author[1]!=author[2]]` is not a univariate XPath because there are two variables ‘author[1]’ and ‘author[2]’ in one predicate.

4.1.3 Subsumption-free XPath

- queries that do not contain redundancies
- some queries can be rewritten such that they become subsumption-free

For example, `//article[(year > 2003) and (year > 2001)]` is not subsumption-free, but it can be rewritten into `//article[year>2003]` which is a subsumption-free XPath query expression.

Figure 4.2 shows the relationship between different XPath languages and the Forward XPath.

![Forward XPath Diagram](image-url)
SwinXSS implements a simplified Forward XPath. It supports multiple atomic univariate predicates and nested predicates within query expressions. However, it does not permit recursively defined elements, “*” and “//”.

In SwinXSS, Forward XPath queries and documents are all represented as trees. We use $u$ to represent the nodes from query expressions, use $v$ to represent the node at instance level. We use $D$ as a representation of a document and $T$ as a tree structure representation of a document. For any $v \in T$, $\text{PATH}(v)$ is the sequence of nodes on the path from the root to $v$. $\text{Child}(v)$ is the set of child elements of the element $v$. $\text{ROOT}(D)$ is the root of document tree $T$. $Q$ is a query tree that consists of all the legal XML node names from an XPath expression. $S$ is the set of all finite length strings of UCS (universal character set) characters.

In Forward XPath language, each node $u$ in the query tree has following properties:

- **AXIS($u$):** Due to the restriction of Forward XPath, AXIS($u$) takes child axis only.
- **LABEL($u$):** Because we do not permit wildcards, LABEL($u$) is from set $Q$.
- **PREDICATE($u$):** PREDICATE($u$) itself is a tree whose internal nodes are tagged by logical, comparison, arithmetic, or functional operators, and whose leaves are tagged by constants from $S$.

---

*Figure 4.3*  Grammar of Simplified Forward XPath

---

<table>
<thead>
<tr>
<th>Path :=</th>
<th>/Step</th>
<th>/Step Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step :=</td>
<td>Element</td>
<td>Element“[&quot;Pred&quot;]”</td>
</tr>
<tr>
<td>Pred :=</td>
<td>Element</td>
<td>Element Oper Const</td>
</tr>
<tr>
<td>Oper :=</td>
<td>“&lt;”</td>
<td>“≤”</td>
</tr>
<tr>
<td>Const is any string from $S$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Func() is any predefined computation.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 4. SwinXSS-A Platform for Query XML Data Streams

Based on all the constraints and definitions above, the table in Figure 4.3 shows the formal grammar of the simplified Forward XPath used in SwinXSS.

4.2 Simplified Forward XPath Query Parser

As mentioned before, in SwinXSS, Forward XPath queries and documents are represented as trees. To decompose the document and get the query tree, we use a simplified Forward XPath query parser. One input for it is XPath expressions. The other input is Schema/DTD, which is the focus of Chapter 5. After the processing by the parser, every node of the query tree is represented as one unit of a Java vector. Each unit consists of several fields: `elementname`, `elementvalue`, `operator`, `noderule`, `leftchild`, `rightchild`, `ancestor`, `vleft`, `vright`. They are useful for matching elements from XML data streams. Their functions are listed below:

- `elementname`: to record the name of an element from the stream.
- `elementvalue`: to record the content belonging to an element from the stream.
- `operator`: to record the operation on the current node if it exists.
- `noderules`: to record the semantic rules applied on the current node if it exists.
- `leftchild`: to record the left child of the current node if it exists.
- `rightchild`: to record the right child of the current node if it exists.
- `ancestor`: to record the parent of the current node if it exists.
- `vleft`: a Boolean value to record the returned value from its left child if it exists.
- `vright`: a Boolean value to record the returned value from its right child if it exists.

For example, the query expression `department/clothes[color="red"]/name` can be mapped into following data structure as shown in Figure 4.4 where $PV$ stands for predicate vector and is set to null if there is no predicate under the query node. We do not show all the fields belonging to the vector. In this example, main path elements ‘`department`’, ‘`clothes`’ and ‘`name`’ are stored in the same vector, and ‘`color`’ is stored in the predicate vector following it.
The detail about the implementation of the query parser, query node and query tree in SwinXSS can be found in `QueryParser.java` and `Querynode.java` classes in the Appendix.

### 4.3 SAX-based Stream Processor

In Figure 4.1, the SAX-based processor sits in the center of the architecture and acts as a core unit in SwinXSS. It takes the streaming XML data and the query tree generated from Forward XPath parser as input, fulfilling the task of matching nodes from query tree with nodes from incoming stream and buffer management, then pushes out the selected data as result. SwinXSS uses SAX parser as the basic tools to divide the streaming XML data into XML grammar units.

We choose SAX because it is the most efficient API. As we mentioned before in Chapter 3, there are other alternative tools or standards that can be chosen to parse the XML document, such as DOM and StAX. However, it is well known that DOM does not perform well when processing large XML documents or XML data streams because DOM needs a tree presentation of the whole document within the memory. For StAX, it is a new API built on SAX or other primitive APIs, which offers the application programmer the flexibility to pull the data from XML document positively without losing the processing efficiency to scan the incoming items. But since StAX is based on the primitive APIs, it just masks the real methods for dealing with some unavoidable problems such as...
buffering to the high level users or applications. So, the processing efficiency and capability problems still exist.

SAX represents XML data as a set of events and pushes them to their registered content handlers through function callbacks. Because SAX parsers push out events in a broadcast fashion, it is efficient for parsing large XML document using limited memory. The fact is that SAX acts as a basic XML processing platform for most of advanced APIs including DOM. So the choice of SAX is wise and reasonable.

We give a brief introduction of our processor constructed on the SAX standard. SAX itself, as a standard stands across different XML parsers. The specific class that implements this interface varies from parser to parser. But they all have important interfaces in common. They are listed as following.

- **XMLReader interface**: An interface implemented by the user to create an object. The object acts as a parser to decompose incoming XML data streams.
- **ContentHandler interface**: An interface implemented by the user to define the methods for different event reactions. The application has to implement all the methods belonging to it.
- **DocumentHandler interface**: An interface has the similar function as “ContentHandler interface”, but there is no need for the application to implement all the methods in it, because the object can be created by default.

In SwinXSS, we use Apache SAX parser tool which uses the common interfaces above to scan then decompose the XML data. When a typical SAX based XML processor like SwinXSS works, an object created by XMLReader reads a document from beginning to end. At the same time, a pointer on the query tree moves back and forth to synchronize the query node matching. During this process, it may encounter start-tags, such as document start or element start tags. All the matching start events will lead to the one step deeper in child axis. It can also encounter end-tags such as document end or element end tags which will conversely lead to pointer one step back in ancestor axis. If text, comments, processing instructions or entities events are encountered, some methods of processing, calculating or operating will be called to make the reactions.
In SAX, the parser tells the client application what it sees as it sees it by invoking methods in an object implementing `ContentHandler` or `DocumentHandler` interfaces. Figure 4.5 demonstrates the relationship between different classes which constitute streaming XML data processor. Within `Main()` function, there is an instance implementing the XMLReader interface which is actually an XML parser. It decomposes XML format data sequence according to the SAX standard defined atomic data units. When decomposing items, an XMLReader object calls the methods in `DocumentHandler` to respond to different predefined SAX events, such as `#PCDATA`, start and end tags etc. Because all the possible reaction programs belonging to a defined atomic event have to be encapsulated within the same function call belonging to that event, it is very clumsy and complicated for a programmer to recognize and select a targeting element in the query tree. So how to distinguish different elements using the same function callback is critical for the SwinXSS query processor.

![XML stream processor implementing SAX interface](image)

**Figure 4.5** XML stream processor implementing SAX interface

As shown in Figure 4.5, regardless what the tag name is, when a start tag is encountered by an XMLReader parser, the `startElement()` function in the `DocumentHandler` is called. It is up to the programmer to implement the functions. If the method body is null, the XMLReader object pushes out the start tag directly. When the `#PCDATA` is encountered, the `characters()` function in
DocumentHandler is invoked to process the values. Because we assume that in any mixed element, the content of that element will always appear as a whole, there is no need to wait for the end tag of the current element. When an end tag is encountered, the endElement() in DocumentHandler is called. Within endElement() function, we often encapsulate all the remaining operations and processing belonging to the current element. So the correspondent algorithm within endElement() is relatively complicated and difficulty to follow. The startDocument() and endDocument() functions are quite similar to startElement() and endElement(). But the startDocument() is mainly used for initializing global variables and external data structures.

We have demonstrated in Figure 4.5 that ContentHandler is a critical interface the client application implements to receive notification of document content. The client application will instantiate a client-specific instance of the ContentHandler interface and register it with the XMLReader. When the reader reads the document, it calls back to the methods in the registered ContentHandler object. The ContentHandler is the only interface and opportunity offered by system to programmer to get the control of different elements for realizing our query system. In Figure 4.6 we list all the indispensable methods which need to be implemented within ContentHandler interface.

```java
void startDocument() throws SAXException;
void endDocument() throws SAXException;
void characters(char[] ch, int start, int length) throws SAXException;
void startElement(String namespaceURI, String localName, String qName,
                  Attributes atts) throws SAXException;
void endElement(String namespaceURI, String localName, String qName)
                  throws SAXException;
```

**Figure 4.6** ContentHandler interface of SAX

58
For detailed information about ContentHandler, it can be found in the Web site of W3C[http://www.w3.org].

With the brief introduction of SAX APIs and the SwinXSS XML stream processor, the formal description for query evaluation can be given:

**Definition 4.1.** A matching of a query node $u$ in $Q$ with a document node $v$ in $T$ of $D$ is a mapping $\Theta$ from $\text{PATH}(u)$ to $\text{PATH}(x)$ (review our previous notations in introduction) with the following properties:

1. **Root match:** $\Theta(\text{ROOT}(Q))=\text{ROOT}(D)$
2. **Axis match:** For any $x \in \text{PATH}(u)$ and according to the restriction of our XPath, $\text{AXIS}(x)=\text{child}$.
3. **Predicate satisfaction:** For all $x \in \text{PATH}(u)$, $\text{PBV}(\text{PEVAL}(rx, \Theta(x)))=\text{true}$, where $rx$ is the root of $\text{PREDICATE}(x)$. $\text{PBV}()$ is the predicate Boolean value function converting values to Boolean, $\text{PEVAL}()$ is the predicate evaluation function, which evaluates predicates against document.
4. **General target match:** $\Theta(u)=x$

A mapping $\Theta$ satisfying all the above conditions without predicates is called structural matching.

**Definition 4.2.** The evaluation of a query tree $Q$ on a document $D$, denoted as $\text{EVAL}(Q, D)$, is a sequence of all document nodes $x$, for which there is a matching with $\text{Release}(Q)$. The order of $\text{Release}(Q)$ nodes are the same as the original document order.

### 4.4 Buffer Management

To achieve the optimized buffer usage, we deploy different buffer usage strategies within $\text{startElement}()$ and $\text{endElement}()$ for different queries. The buffer manager will coordinate all those strategis according to the information from query nodes of the query tree.
Figure 4.7 Basic buffer management in SwinXSS

As shown in Figure 4.7, the buffer manager maintains a series of Boolean flags. The combined validation of different flags can lead to the execution of different query strategies within `ContentHandler()`. For example, when a matched tag ‘d’ is encountered, as shown by the pointer pointing d in the query tree, the buffer checks the flag series and finds that flag i and flag j are valid in this example. The combined effect of these two flags will lead to an execution of a strategy within each event handler.

With the brief introduction of our system buffer management mechanism, we present the detailed algorithms for our query processor in Section 4.5. The detailed description for flags is also given.

4.5 Query Evaluation Algorithm with Concurrency Lower Bound

In SwinXSS, the leftmost path of a query tree is called the main path of the query and the leaf node in this path is called the query output node. Other paths of the query tree are called the predicate paths. To compute the concurrency lower bound, we develop a best effort algorithm that delivers or discards query output nodes whenever all the predicates defined in the query are evaluated such that the number of live elements of the output node is the lowest. The algorithm works by processing the stream of SAX startElement and endElement events for
startElement(e) and endElement(e) where \( e \) is the element tag event. We simply treat it as an element. In the real implementation, we also have content event handler that deals with the buffering and outputting of the content of the output node elements.

In the algorithm, we use a stack to buffer the elements for the query output node and other main path nodes that are necessary for the query evaluation. stackNode is used to record the query node that starts to use the stack and entryNode the parent node of stackNode, predicateNode is used to record the root node of any predicate subtree. The initial values of these variables are all set to “null”. outputNode is used to record the query output node. stackFlag marks that the stack is being used and PredicateFlag marks that predicates are being evaluated, both having “false” as the initial value. Each query node qNode may have a parent, leftChild, and rightChild and use leftCondition and rightCondition to record the predicate evaluation results from its leftChild and rightChild, respectively. The initial value of qNode takes the root node of the query tree as leftChild and null as rightChild.

In startElement(), we first process start-tags of those query nodes in the main path up to stackNode which has predicates to be evaluated (Lines 1-10). In case there is no predicate at all in the query tree, output the elements for outputNode immediately (Line 4). If the start-tag matches a node in the main path below entryNode, it is pushed in the stack for predicate evaluation later (Lines 13-17).

To calculate the concurrency lower bound defined in [6], we only count the query output node. We use concur to record the concurrency of current time step and concurLB to record the maximum concurrency up to now. After the stream is processed, concurLB yields the document concurrency from which the lower bound can be obtained (Line 16). If the start-tag matches a predicate node, preparations will be made (Lines 18-21) and predicate evaluation will follow (Lines 24-25).

**Function startElement(e)**

```java
1. if (!stackFlag) { // stackFlag = false
2.     if (qNode.leftChild == e) {
```

Chapter 4. SwinXSS-A Platform for Query XML Data Streams
3. qNode = qNode.getLeftChild();
4. if (qNode = outputNode) output(e);
5. if (qNode.leftChild.rightChild != null) {
  
6.   stackFlag = true; entryNode = qNode;
7.   stackNode = qNode.getLeftChild();
8. }
9.
10. }
11. else { // stackFlag = true
12. if (!predicateFlag) { // predicateFlag = false
13.   if (qNode.leftChild = e) {
14.     Stack.push(e);
15.     qNode = qNode.getLeftChild();
16.     if (qNode = outputNode) {concur++; if (concur > concurLB)
17.       concurLB++;}
18.   }
19.   if (qNode.rightChild = e) {
20.     qNode = qNode.getRightChild();
21.   }
22. }
23. else { // predicateFlag = true
24.   if(qNode.leftChild = e) qNode = qNode.getLeftChild();
25.   if(qNode.rightChild = e) qNode = qNode.getRightChild();
26. }
27. }

In endElement(), Line 2 is used to process an end-tag that matches a node above stackNode. Lines 4-31 are used to process an end-tag that matches a node below entryNode in the main path while Lines 32-40 are used to process an end-tag that matches a node in a predicate path. Given a matching node qNode, the function checkLeftRight(qNode) checks whether both its rightCondition and leftCondition are true or not (Line 5). For the case of true, we check if qNode is the stackNode, and if so, we empty the stack and output elements if it matches
outputNode and adjust current concurrency accordingly (Lines 7-12). We take an
eager predicate evaluation approach that measures the concurrency lower
bound exactly. Whenever qNode is evaluated to be true from both leftChild and
rightChild in Line 5, we check if all its ancestors up to stackNode are also
evaluated to be true from its rightChild by the function checkRight(). If so, we
immediately pop up the stack to the element that matches qNode, output all
elements that match outputNode and adjust the current concurrency as well
(Lines 16-22). Similarly, whenever qNode is evaluated to be false from either
leftChild or rightChild, we also immediately pop up the stack to the element that
matches qNode, discard all elements in the stack including those match
outputNode, and adjust the current concurrency (Line 27). In other words, we
keep concur and hence concurLB as low as possible and this algorithm reflects
the concurrency lower bound calculation.

**Function endElement(e)**

```
1. if (qNode = e) {
2.  if (!stackFlag) {qNode = qNode.getParent(); qNode.leftCondition = true;}
3.  else {// stackFlag = true
4.    if (!predicateFlag) { // predicateFlag = false
5.      checkedResult = checkLeftRight(qNode); reset(qNode);
6.      if (checkedResult) { // both leftCondition and rightCondition are true
7.        if (qNode = stackNode) {
8.          while (Stack.size!=0) {
9.            t = Stack.pop();
10.           if (t = outputNode) {output(t); concur--};
11.          }
12.        }
13.        if (qNode = entryNode) { stackNode = null; stackFlag =false;};
14.        qNode = qNode.getParent();
15.        qNode.leftCondition = true;
16.        cNode = qNode;
17.        while (checkRight(cNode)) {
18.          if (cNode=stackNode) {
19.            do {t=Stack.pop(); if (t=outputNode) {output(t); concur--}} until
```
Chapter 4. SwinXSS-A Platform for Query XML Data Streams

(t=e);
20. }
21. else cNode=cNode.getParent();
22. }
23. }
24. else {// either leftCondition or rightCondition is false
25. if (qNode = entryNode) { entryNode=null; stackFlag=false;}
26. else {
27. do {t = Stack.pop(); if (t = outputNode) concur--} until (t=e);
28. qNode = qNode.getParent();
29. }
30. }
31. }
32. else {// predicateFlag = true
33. checkedResult=checkLeftRight(qNode); reset(qNode);
34. if (checkedResult ∧ (predicateNode = e)) {
35. predicateFlag = false; predicateNode = null;
36. }
37. qNode = qNode.getParent();
38. if (qNode.leftChild = e) qNode.leftCondition = true;
39. if (qNode.rightChild = e) qNode.rightCondition = true;
40. }
41. }
42. }

4.6 Summary

In this chapter, we introduce an XML data stream query system SwinXSS that is based on SAX, then, all the components. After that, the algorithm for the query system is given. We explain that the algorithm achieves the performance with the currency lower bound.
Chapter 5: Semantic Query Optimization over XML Data Streams

Given that buffering may cause a major memory bottleneck and space complexity measured as concurrency lower bound has been theoretically proved as the limit that any algorithm can achieve, can we by any means break this bound? In this section, we aim to give a positive answer to the question by exploring semantic information and use it for buffer reduction.

5.1 Analysis of Concurrency Lower Bound

From the algorithm presented in Chapter 4, we can define three states for a query output element being processed: live, selected, and discarded. Selected and discarded states are certain for the element to be outputted or ignored, respectively while a live state is uncertain and buffer is required for the element.
Chapter 5. Semantic Query Optimization over XML Data Streams

with a live state. From the algorithm, we keep the number of live output elements as low as possible so the concurrency lower bound is achieved.

We denote the document concurrency \( \text{CONCUR}(D, Q) = f(l) \) where \( l \) is the deepest layer an output element may appear in the stream \( D \). Suppose that at a certain layer \( k \), the maximum number of the elements which have the same parent is \( M(k) \). For example, in Figure 5.1 the \( M(k) \) at layer 2 is 3, and 2 at layer 3. So to any document \( D \), at any layer \( k \), the total number of nodes will not exceed \( t(k) = \prod_{x=k}^{1} M(x) \) where \( x \) is the layer variable. If all the elements are all live in our example, the total live element at layer \( k \) will not exceed \( t(k) = \prod_{x=k}^{1} M(x) \). So to evaluate the query against the whole XML document, if there exists a predicate at layer 1, we get \( \text{CONCUR}(D, Q) = f(l) \leq t(1) + t(2) + \ldots + t(l) \). So we can prove \( \text{CONCUR}(D, Q) \leq \sum_{k=1}^{l} \prod_{x=k}^{1} M(x) \).

According to the definition of the \( f(l) \), in order to keep the query being valid, in the whole XML document, there must exist minimum one path reaching the leaf node pointed by the XPath query. Based on the Forward XPath, the minimum number of live elements is \( \sum_{x=1}^{i} \prod_{k=1}^{x} \). If the ‘//’ queries and recursive elements are not taken into consideration, we can get \( f(l) = 1 \) or \( \text{CONCUR}(D, Q) = 1 \). So we can give the following range for the document concurrency:

\[
\sum_{k=1}^{l} \prod_{x=k}^{1} \leq f(l) \leq \sum_{k=1}^{l} \prod_{x=k}^{1} \text{MAX}(k)
\]

Note that the right side of the above formula is coincident with the space consumption of linear XPath without predicates, whose important character is that the states of all output elements can be immediately determined as either selected or discarded upon their arrivals. In such case, no buffer is needed at all. \( \text{CONCUR}(D, Q) \) is calculated in a way that whenever all related predicates for
an output element have been evaluated, action is taken immediately to either 
*output* or *discard* the element.

However, the output element has to be buffered if those predicates are not 
yet evaluated so the live state of the element can not be converted to either 
*selected* or *discarded* at the time. If we can take the advantage of semantic 
information and make the evaluation of some predicates early, we can change 
the *live* state of the output element early. In other words, we can change the 
query into linear one and break the lower bound!

### 5.2 Semantic Rules for Buffer Reductions

With the above analysis as guidelines, we explore useful constraints from 
schema in a DTD or XML Schema and design semantic rules to use them for 
buffer reduction.

**Rule 1: Predicate After Rule**

It is easy to find in a schema the appearing order between those nodes in the 
main path including the output node and those in predicates. Actually this 
information is especially important because we want to evaluate the predicates 
early so that the elements for the output node can go through early. Given a 
node \( v \) in the main path of a query tree, if from schema we know that the 
elements of its right child \( p \) always arrive after the elements of its left child \( a \), 
we may apply the *Predicate After Rule* denoted as 
PREDAFTER(Child(\( v \))=a,Child(\( v \))=p) for buffer reduction. This rule states that 
each \( a \) element arrives before any \( p \) element. If there exists a constraint \( f(a,p) \) 
which becomes true after the arrival of certain number of \( a \) elements and this 
change triggers that the predicate on \( p \) also becomes true, then the previously 
buffered \( a \) elements under \( v \) and subsequently arriving \( a \) elements can be 
outputted immediately. The processing procedure is shown in Figure 5.2.
For example, in a stock market, ordinary users can open as many as 5 windows to observe the market, but a VIP user can open as many windows as he or she wants. If \text{PREDAFTER}((\text{Child}(\text{user}) = \text{window}), \text{Child}(\text{user}) = \text{VIP}) and the query is \text{/market/user[\text{VIP}]/window}, then once the 6\text{th} window arrives for a user, we can immediately output the buffered 5 windows and the current window and the subsequent windows for the user before the start-tag of \text{VIP} arrives. However, the lower bound algorithm will have to wait until either the start-tag of \text{VIP} or \text{user} arrives.

\begin{equation*}
f(a,p) = \text{T}
\end{equation*}

**Figure 5.2** Influence after deployment of \textit{Predicate After Rule}

\textbf{Rule 2: Predicate Ahead Rule}

Similarly for a node \(v\) in the main path of a query tree, if from schema we know that its right child \(p\) is before its left child \(a\), we may apply the \textit{Predicate Ahead Rule} denoted as \text{PREDAHEAD}((\text{Child}(v) = a), \text{Child}(v) = p). This rule is especially important to immediately dump \textit{live} output elements which will eventually be discarded. If with respect to a certain query node \(v\), this rule can be satisfied, the buffer usage is avoidable as demonstrated in Figure 5.3.

For the previous example, if the query is the same but the rule is changed from \text{PREDAFTER} to \text{PREDAHEAD}, then we do not need to buffer window elements at all. If \text{VIP} does appear for a user, all \text{window} elements arriving later will be immediately outputted; otherwise, they will be discarded. For the latter,
the lower bound algorithm has to buffer all the window elements until the end-tag of user element arrives.

Figure 5.3  Influence after deployment of Predicate Ahead Rule

**Rule 3: Maximum Cardinality Rule**

If knowing that one $v$ element has at most $n$ elements for child node $e$, we may apply the Maximum Cardinality Rule denoted as MAXI(Child($v$)=e, $n$). XML Schema provides $\text{maxOccurs}$ to specify this information. In Figure 5.4, with the appearance of $\text{maxOccurs}$, the buffer will be released to save the space consumption.

Figure 5.4  Influence after deployment of Maximum Cardinality Rule
Chapter 5. Semantic Query Optimization over XML Data Streams

If we have \( \text{MAXI}(\text{Child}(\text{user})=\text{interest}, 4) \) and the query is \( /\text{market}/\text{user}[\text{interest}='\text{golf'}]/\text{stock} \). The 4\textsuperscript{th} end-tag of interest will enable us to discard all buffered window elements and future arriving window elements of the user immediately. However, this cannot be achieved by the lower bound algorithm.

**Rule 4: Co-exist Rule**

From cardinality constraints defined in a schema, we may infer the coexistence of a pair of elements \( a \) and \( b \) under \( v \), denoted as \( \text{COEX}(\text{Child}(v)=a, \text{Child}(v)=b) \). There is no need for any processing of streaming XML data for the deployment of this rule because the semantic equivalence is used to rewrite the query before the execution of original query plan which lead to the direct output of any element ‘a’ as shown in Figure 5.5.

\[
\begin{array}{c}
\text{Figure 5.5 Influence after deployment of Co-exist Rule} \\
\end{array}
\]

For example, if we have \( \text{COEX}(\text{child}(\text{user})=\text{VIP}, \text{child}(\text{user})=\text{vroom}) \), and the query \( /\text{market}/\text{user}[\text{VIP}]/\text{vroom} \), we can immediately output \( \text{vroom} \) elements with no need to wait and check \( \text{VIP} \). Similarly we may have the exclusive rule \( \text{EXC}(\text{Child}(v)=a, \text{Child}(v)=b) \), which means that either \( a \) or \( b \) is a child element of \( v \) but not both. The query in the form of \( /v[a]/b \) will not output any \( b \).
5.3 Influence from Rules Towards Buffer Management

It has been described in Section 4.4 that without the semantic rules, the buffer manager finds the efficient query plan using the information mainly from the query tree. But if there is semantic information which can be referred, the situation can be more complicated. The main purpose for the deployment of our semantic rules is to decrease the buffer usage. Actually, with the reference to semantic information, all the reduction on buffer will be realized through the buffer manager which coordinates the semantic rules and processing blocks flags together.

As shown, in Figure 5.6, the buffer manager not only maintains a flag sequence but also a rule sequence. Each unit in the rule sequence holds a Boolean flag that indicates whether the rule is applicable. The buffer manager will combine the Boolean flags and rules to select the most optimized execution strategy in the `ContentHandler()`.

![Diagram](image_url)

Figure 5.6  Influence of semantic rules to buffer management

In the example of Figure 5.6, the matching of a certain ‘d’ tag in the query tree triggers rule l and flag j. The incoming document can also change the Boolean value in both the rule sequence and the flag sequence. For example, a start tag event on a certain element ‘d’ may trigger rule k. Consequently, the combined effect of valid rule k and rule l triggers flag i. Under this
circumstance, the buffer manager fulfils the task for coordinating all the 2 flags and 2 rules together to select an optimal strategy within each event handler.

The details about above Boolean flags and rules also can be found in Section 4.5 and Section 5.4 respectively.

### 5.4 Incorporation of Semantic Rules into Lower Bound Algorithm

In the previous 2 sections, we introduce our SQO rules and analyse their influence on buffer management. In this section, we show how to incorporate them into our basic concurrency lower bound algorithm.

If the PREDAFTER(Child(v)=a, Child(v)=p) where v, a, and p correspond to qNode, qNode.leftChild, and qNode.rightChild, respectively, and there exists constraint $f(a,p)$ between a and p, we add Lines a-n between Line 13 and Line 14 in the `startElement()`. If $f(a,p)$ becomes true after current a element arrives (Line a), we infer that the predicate will be evaluated to be true and thus no need to be evaluated (Line b). We then check all `rightChild` of those nodes up to `stackNode` and see if they are all true (Lines c-g). If so, we pop up the stack up to `qNode` and output elements that match `outputNode` and adjust the current concurrency (Lines h-m). After that we continue with the processing of the arrived element.

**Changes in startElement() for Predicate After Rule**

```
a) If (PREDAFTER(qNode.leftChild, qNode.rightChild) ∧
    f(qNode.leftChild, qNode.rightChild)) {

b) qNode.rightCondition = true;
c) cNode = qNode;
d) while !(cNode=stackNode) {
e) cNode=cNode.getParent();
f) if !checkRight(cNode) skip;
g) }

h) if (checkRight(cNode) {
i) while (!Stack.top()==qNode) {
   j) t=Stack.pop();
```

72
Chapter 5.  Semantic Query Optimization over XML Data Streams

k) if (t = outputNode) {output(t); concur--};
l)
m)
n)

If PREDAHEAD(Child(v)=p,Child(v)=a) where v, a and p correspond to qNode, qNode.leftChild and qNode.rightChild, respectively, we add Lines o–r also between Line 13 and Line 14 in startElement(). The arrival of the first start-tag of a symbolizes the end of all p elements. If we know all p elements are evaluated to be false by checkRight(), we pop up the current top node in the stack which is qNode. Then we set qNode to its parent node to bypass the processing of the subtree rooted with qNode. If the checking in Line o fails, we continue with Lines 14-16 in the original algorithm (Line s).

Changes in endElement() for Predicate After Rule

o) if (PREDAHEAD(qNode.leftChild, qNode.rightChild) ∧ !checkRight (qNode))
   {
   p) Stack.pop();
   q) qNode=qNode.getParent();
   r) }
   s) else {Lines 14 – 16 in (endElement() in section 4.5)}

The treatment of the Maximum Cardinality rule is similar to that of the Predicate After Rule. Instead of checking \( f(a,p) \) in Line a, we count the number of arriving a elements by carrying out the count() function to see if it reaches the maximum cardinality, \( \text{maxOccurs} \).

Changes in startElement() for Maximum Cardinality Rule

t) If (MAXI(qNode.rightChild, n) ∧ (count(e)=maxOccurs)) {
  u) qNode.rightCondition = true;
  v) cNode = qNode;
  w) while !(cNode=stackNode) {
  x) cNode=cNode.getParent();
Chapter 5. Semantic Query Optimization over XML Data Streams

The treatment of the Co-exist rule is also similar. Unlike that of the Maximum Cardinality rule, we do not need to check extra condition. The correspondent can be seen as follows.

Changes in startElement() for Co-exist Rule

5.5 Example Run

We demonstrate the optimized algorithm by an example. Experiment results will be shown in Chapter 7. As shown in Figure 5.7, we have $Q = a[p]/b[m|x]$
const 1]/n < const 2]/c, which could be normalized into query expression \( Q = a[p]/b[m/x > Const1 \&\& m/n < Const2]/c \), and our algorithm treats above two expressions as equal queries. In brief, a recursive nested predicate in a query can be computed as non-recursive one. The tree structure representation of the query can be seen in Figure 6(a). The light nodes are elements on the main path, and dark nodes are predicate nodes. The incoming instance document sequence \( D \) is noted as SAX events in Figure 5.7(b), the query will select \( \Theta(u) = v \) [refer to Chapter 4.3] from \( D \). The semantic information belonging to query node \( b \in Q \) is \( <!ELEMENT b(m*, c+) > \), of which the structure and predicate sequence can determine the optimization. From Figure 5.7(a), we know node \( b \in Q \) carries a predicate. Another semantic information for query node \( a \in Q \) is \( <!ELEMENT a(b+, p+) > \), which clarifies that the appearance of \( b \in D \) or \( p \in D \) will determine the appearance of each other.

In Figure 5.7(b), we give an example XML stream document for the query algorithm demonstration. The root element of the whole document is element \( a \in D \). From the root, along the arrow direction, the incoming XML stream comes to an end tag ‘/a’. With the reference to this flow, in Figure 5.7(c) and Figure 5.7(d) we demonstrate the detailed steps of buffer operation.

Without the help of semantic information, the basic algorithm will store most of \( c \in D \), which can be seen in buffer table of Figure 5.7(c). The basic algorithm is developed according to the concurrency lower bound. So all the live elements \( c \in D \) belonging to the sub-tree under the current element \( a \in D \) have to be stored. The reason is that without the confirmation of arriving of \( p \in D \), the query processor can not determine whether all the buffered \( c1, c2 \in D \) elements are qualified for the query expression. Predicate \( p \) stands at the high level of the query tree, even the predicate belonging to element \( b \in D \) is qualified, the state of elements \( c1 \in D \) and \( c2 \in D \) will still keep ‘live’ unless \( p \) is also evaluated. With respect to predicate expression belonging to query node \( b \in Q \), because the number of element \( m \in D \) is not clear, if there is no arrival of the end tag ‘</b>’ and the qualification information of the predicate belonging to query node \( b \in Q \), elements \( c1 \in D \) and \( c2 \in D \) need to be buffered. The processing of predicate under \( b \in D \) is as follows: as soon as an element \( m \in D \)
is encountered, the expected elements will be set to $x$ and $n$. If the next incoming element is one of the expected elements, a Boolean value will be set for it. The similar explanation can be used to describe the processing of predicate under element $a \in D$.

In Figure 5.7(a), DTD <!ELEMENT $b(m^+, c^+)$> analysis shows, to any $b \in D$, all possible elements $m \in D$ need to arrive ahead all the elements $c \in D$. To any $a \in D$, the analysis of <!ELEMENT $a(b^+, p^+)$> shows that when the query processor encounters an element $b \in D$, there must exist a qualified element $p \in D$. The above analysis shows that we can treat the match of query node $a \in Q$ as a none predicate query evaluation.

![Diagram of buffer management optimization](image)

**Figure 5.7** Buffer management optimization

With the above semantic information, we can optimize the usage of buffer to a linear level. As shown in Figure 5.7(d), because the semantic information helps the query processor change the query with predicate $p \in Q$ into linear one,
Chapter 5. *Semantic Query Optimization over XML Data Streams*

it will not cause any buffer usage here. Consequently, with the arrival of \(<c_1>\), if under current \(b \in D\) there is still no element \(m \in D\) i.e., satisfying the predicate, the whole buffer could be emptied because of no suitable predicate for current element \(b \in D\). On the other hand, if under current \(b \in D\) there is qualified element \(m \in D\) satisfying the predicate, all the buffered ‘c1’ or ‘c2’ will be output. By the comparison of buffers in Figure 5.7(c) and Figure 5.7(d), the algorithm using information from DTD cuts the buffer space significantly.

According to the theoretical concurrency lower bound introduced in Chapter 3, \(\Omega(\text{CONCUR}(D, Q))\) bits of space is unavoidable. Normally, the CONCUR\((D, Q)\) is the repetitive frequency of element \(c \in D\). In this example, the CONCUR\((D, Q)=n\) where \(n\) is the maximum number of \(c\) under any \(b \in D\) and the lower bound of the algorithm is \(\Omega(n)\). But when the DTD information is used, the buffer space complexity can be dramatically reduced. In this example, the actual CONCUR\((D, Q)\) of elements in the buffer during the evaluation becomes a constant 1 which is the linear query lower bound.

### 5.6 Summary

In this chapter, the analysis of the theoretic lower bound is given. Based on that, we introduce a criterion for finding the semantic rules for SQO over XML data streams. We design 4 types of semantic rules and incorporate these rules into the basic algorithm in Chapter 4. Finally, an example run is used to demonstrate the semantic optimization process according to different rules.
Chapter 6: Processing and Optimization for Block Operators

As we have introduced in previous chapters, a block query operator is unable to produce the first tuple of its output until the entire input has been seen. Sorting is a blocking operator, so are aggregation operators including SUM, COUNT, MIN, MAX, and AVG. The processing of block operators is a difficult task in data stream querying processing.

Under XML data streams model, we model the stream data as SAX events. This means that when it is scanned, it is discarded unless it is buffered. This makes it hard to satisfy the requirement of a block operation.

6.1 Block Operation Scenarios

As shown in Figure 6.1, the query expression at the top with ‘Sum()’ function returns the total score for each ‘student’ belonging to a certain ‘department’. To get the query result, the query processor has to scan all the ‘course’ elements under a ‘student’ element and add them together. In order to achieve that, we divide the processing into two steps:

1. To group the ‘course’ elements according to each ‘student’ element.
2. To accumulate all the ‘course’ element values within the group.

To complete the block operation in this example, the query engine needs to distinguish every different sub-tree of a ‘student’ element, then the
aggregation of ‘course’ will be executed within that sub-tree. It needs to scan a whole sub-tree to get the total value.

**Figure 6.1** Block operation query scenario

### 6.2 Block Operation over XML Data Streams

In general, block operations can be evaluated in two ways. Firstly, the evaluation starts at the beginning of the XML data stream. Secondly, the evaluation starts in the middle of the XML data stream. For example, as shown in Figure 6.2, when the XML stream runs to the ‘course’ element with its name as ‘c2’, the processor begins to evaluate an *ad hoc* query. But the query needs to use the value of ‘course’ element to calculate the summary of all the ‘score’ elements under the same ‘student’ element. The problem lies in that the ‘score’ elements of some ‘course’ elements may be passed by. The popular answer to this problem is to use synopsis.
6.2.1 Aggregate Functions

Aggregate functions can be processed by adopting certain query strategies without buffering all the elements under that block. To deal with this type of operation, we only need to maintain a couple of variables to get the final result step by step. This type of operators include ‘MIN’, ‘MAX’, ‘COUNT’, ‘SUM’, ‘AVERAGE’ etc.
For example, in Figure 6.3 there is an XPath query expression, \( \text{Departement/Student}[\text{SUM(Score)} > 160]/\text{Student}\_ID \). Every time when the Query processor encounters a start tag of ‘student’ element, it will initiate and maintain a variable in the memory to record total score of each student. With the arrival of the end tag of ‘student’ element, the query processor will compare the variable with 160. If the value is bigger than 160, it sets the predicate to ‘TRUE’; otherwise, it sets the predicate to ‘FALSE’.

### 6.2.2 Update Operations

To explain the update operations over XML data streams, we use an example in Figure 6.4. The document has ‘department’ as its root. The only child node of ‘department’ is ‘student’ which in turn has ‘score’, ‘rank’ and ‘sID’ as its child elements. The user defined update operation requires to rewrite the ‘Rank’ elements according to the average score for each student.

![Diagram](image)

**Figure 6.4** The influence from document towards update operation
In the document shown in Figure 6.4(a), because all of the ‘Rank’ elements arrive after all the ‘Score’ elements under the same ‘student’ in the whole XML document tree, the update processing is easy. The query processing only needs to follow the sequence of XML stream and use a variable to record the summary of all the ‘score’ elements belonging to the same student. Then we get the average value belonging to each student from that variable and compare it with the Rank table at the top of Figure 6.4. Finally, the following ‘Rank’ element belonging to that ‘student’ will be modified.

But things become complicated when the incoming stream document is not so “regular”. In Figure 6.4(b) the arriving order of ‘Rank’ element is uncertain. For example, the ‘Rank’ elements belonging to ‘S23656’ and ‘S20996’ appear in different positions in the sub-trees where they are located. For the first ‘student’ with its ‘sID’=S23656 in Figure 6.4(b), the ‘Rank’ element arrives before all the ‘score’ elements. To update it, the query processor has to wait until the arrival of all the three ‘score’ elements for the average value done.

6.3 Semantic Query Optimization for Block Operations

Given that the evaluation of block operations over XML data streams are time consuming, can we explore semantic information to optimize the processing of them? We address this in the rest of this section, as its first attempt.

In Chapter 5, we have given a main criterion for finding our optimization rules on querying XML data streams. It generally tells that if we can take advantage of semantic information and make the evaluation of predicates early, we can change the live state of the output element early to reduce buffer cost. Whether can it be used to find the rules for optimizing the queries carrying block operations?

Figure 6.5 shows a query with ‘Average()’ operation for selecting all the students from qualified departments. In the document shown in Figure 6.5, all the predicate elements ‘Rating’ are read first. After that, the result of block operation will be evaluated to determine the Boolean value of the predicate. If the predicate is ‘True’, it has the result as the linear query /University/
Department/ Student. Can we evaluate the predicate as early as possible? The answer is “yes” if some semantic information can be utilized. The following rule is one of the examples.

**Rule 5: Predicate Range Rule**

In a schema the data ranges of some types of elements can be set. With these ranges, sometimes we can predict the results of some predicates. For a node $u$ in the main path of a query tree, if the schema states that the elements of its right child $p$ can only have a value ranging from $(m1~n1)$ and there is a predicate on a block operation function $\text{func}(p)$ on $p$. Sometimes, there is no need to see all $p$ in the document $D$ in order to evaluate the predicate. Because the data range of $p$ can determine the $\text{func}(p)$ in advance.

We denote the *Predicate Range Rule* denoted as $\text{PREDRANGE}((\text{FUNC}((\text{Child}(v)=p)), \Theta))$. This rule states that there is a predicate on a block function ‘FUNC()’. If this predicate can be evaluated ‘True’ after a certain $p$ arrives, we can release the buffer immediately.

![Figure 6.5 Quantifier requirement of Block Operation](image-url)
Figure 6.6  Example for Predicate Range Rule application

For example, in Figure 6.6, the XML Schema has given the data range for the rating that each marker can give to a department. The query returns all the students of those departments those have the average rating of no less than 4. If there is no Schema information, the query engine has to wait for the incoming of rating of ‘5’ and ‘4’ as shown in Figure 6.6 to determine the process for all the buffered ‘student’ elements. Whereas, with the Schema information, after the reading of the second rating ‘1’, the query can judge that the maximum average of block operation \( \text{Average}(\text{rating}) \geq 4 \) is impossible.

6.4 Summary

In this chapter, we use several application scenarios to explain the concept and requirements of a block operation. Then, we explain several types of block operations and how they are processed over XML data streams. Due to the cost of block operation processing, we further discuss the possibility of applying SQO technique to reduce the space complexity for block operations over XML data streams.
Chapter 7: System Performance and Empirical Study

Figure 7.1  XMark document structure fragment
Both the concurrency lower bound algorithm and the algorithms incorporating the semantic rules are implemented in SwinXSS using Java. In this chapter, we describe our experiments which show the system performance. Experiments are conducted on an Intel P4 3GHz PC with 512 MB memory.

7.1 XMark Document

We generate experimental documents using the XMark document generating tool\(^1\) with the XML DTD auction.dtd as input. The partial tree structures of the testing documents are shown in Figure 7.1 and 7.2. In auction.dtd, the maximum depth of the documents is 9 steps without taking recursive structures into consideration. This is useful to test the buffer usage effectively. The width of the document is also sufficient and structure is variable.

\(^1\) http://monetdb.cwi.nl/xml/index.html

![Figure 7.2 XMark document structure fragment](image-url)
There exist 6 types of child nodes under the root node ‘site’, they are all indispensable. Figure 7.1 shows the detail of ‘region’ node. The dash lines specify the exclusive relationship between elements. In the example, only one continent can be chosen. Each element for a continent has a similar structure for auction item as the ‘africa’ element.

Figure 7.2 shows the detail of the ‘people’ node. Some correlation information is defined between elements, e.g., when the quantity of ‘watch’ elements in the real document exceeds a certain value, the ‘person’s descendant element ‘business’ must exist. Here all ‘watch’ elements are always ahead of the ‘business’ element for any ‘person’ element.

### 7.2 Query Design

<table>
<thead>
<tr>
<th>Query</th>
<th>Forward XPath Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>/site/regions/asia/item/name</td>
</tr>
<tr>
<td>Q2</td>
<td>/site/people/person/profile/business/watches/watch</td>
</tr>
<tr>
<td>Q3</td>
<td>/site/regions/africa/item/mailbox/mail[date&gt;2002]/text/keyword</td>
</tr>
<tr>
<td>Q4</td>
<td>/site/regions/africa/item/incategory=&quot;category18&quot;/mailbox/mail/text/keyword</td>
</tr>
<tr>
<td>Q5</td>
<td>/site/regions/africa/item/shipping/description/parlist/listitem/text/keyword</td>
</tr>
<tr>
<td>Q6</td>
<td>/site/people/regions/africa/item/description/parlist/listitem/text/keyword/mailbox/mail[date&gt;2001]/text/keyword</td>
</tr>
</tbody>
</table>

**Figure 7.3** Test queries

Taking both the document and the semantic information into consideration, we then design 6 queries in Figure 7.3 that are used to test the effectiveness of each semantic rule we designed in Chapter 5. We use bold font to emphasize the points where different semantic rules can be applied. Q1 is a query without predicates. Q2 and Q3 are designed to test Predicate After and Predicate Ahead
rules, while Q4 and Q5 are for Maximum Cardinality and Co_exist rules. The Q6 is designed to verify the combined effects when all the semantic rules are applied collectively. For each of Q2-Q4, we assume that the corresponding rule is applicable. Furthermore, \( \text{count}(\text{watch}) > 6 \rightarrow \text{business} \) holds for Q2 and \( \text{MAXI}(\text{child}(\text{item}) = \text{incategory}, 3) \) for Q4.

### 7.3 Comparison of Maximum Buffer Scale

All of the above queries are evaluated on the generated 1GB and 2GB document, respectively. This experiment targets to compare the peak values of buffer scales before and after the deployment of semantic rules. Figure 7.4 (a) and Figure 7.4 (b) show the results of evaluating Q1 – Q6, for the 1G and 2G documents. The 6 bars from left to right stand for the results for the lower bound algorithm, the individual algorithms for Rules 1-4, and the algorithm for applying all the rules for combined optimization, respectively.

For Q1, we can see that the document concurrency for all algorithms is 1 for both 1G and 2G documents because there is no predicate. For Q3 and Q5, we can see that the marvellous effect of applying the Predicate Ahead rule and the Co-exist rule, which make the document concurrency reduced to 1 for both 1G and 2G documents while the lower bound algorithm can achieve 36 and 24 in 1G document for Q3 and Q5, and 56 and 33 in 2G document for Q3 and Q5. For Q2, the document concurrency only depends on the constraint between person’s two descendant child elements business and watch, i.e., \( \text{count}(\text{watch}) > 6 \rightarrow \text{business} \). Thus, the document concurrency is 7 for both 1G and 2G documents. For Q4, the reduction effect is highly related to the specific content of document, which can be seen by comparing the document concurrency achieved for 1G document with that for 2G document. The former is better than the latter. Obviously, for Q2-Q4, the algorithm for combined optimization takes the best document concurrency of those algorithms that apply the individual rules. The collective effect of buffer reduction is demonstrated in Q6, where each algorithm that applies individual rule does no reduction while the combined optimization algorithm performs perfectly. This is because there are three predicates in Q6. The selected state of an output
Chapter 7. System Performance and Empirical Study

element *keyword* depends on three ancestor elements *site*, *item* and *mail*. The application of each individual rule may not pre-determine the predicates of all three elements.

![Figure 7.4](image_url)  Maximum buffer scale for 1GB and 2GB XML dataset

**Figure 7.4** Using the semantic information for Concur lines breaking

Figure 7.4 shows that if there are predicates in a query and useful semantic information, the lower bound can be broken by the algorithms with semantic rules. We can see that the combined optimization algorithm always
outperforms the lower bound algorithm. The experimental result is consistent with our expectation.

We extract the detailed data from Figure 7.4 about concurrency and the data of combined optimization to make a comparison. Figure 7.5 shows if there exist predicates in a query and there exist applicable semantic rules, the lines with semantic query optimization are significantly below the lines with the currency lower bound $\text{CONCUR}(D, Q)$. Consequently, we empirically prove the break-through of the limitation of the concurrency lower bound by exploring semantic information.

### 7.4 Comparison of Response Time

![Figure 7.6](image)

**Figure 7.6** Response time for 1GB and 2GB XML dataset

Figure 7.6 shows the experimental results on execution time needed for evaluating Q1-Q6 using 6 different algorithms. Because of the saving in buffer processing, the algorithms using semantic rules outperform the lower bound algorithm. From the figure, we can find that the combined optimization algorithm and the algorithms that apply the Predicate Ahead and Co-exist rules perform better than the algorithms that apply the Predicate After rule and Maximum Cardinality rules because the former three algorithms use fewer buffer processing time. In general, the reduction in response time is less than...
the reduction in buffer consumption because each algorithm has to scan the whole document and the only saving in time comes from the saving in buffer processing.

7.5 Summary

In this chapter, we first describe the important fragments of an XMark document and design several queries which are used in our experiments. Then we show our experiment conducted using the basic concurrency lower bound algorithm and the algorithm with semantic optimization rules implemented in SwinXSS. Both maximum buffer scale and response time are compared. Our experiment results show that SwinXSS significantly outperforms the concurrency lower bounds with the help of our defined semantic rules.
Chapter 8: Conclusions and Future Work

8.1 Conclusion

The research described in this thesis focuses on the XPath query evaluation against XML data streams and its optimization techniques based on semantic information.

To efficiently process queries on XML data streams, we identify its characteristics and propose an effective solution in Chapter 4. The problem of XPath query evaluation against XML data (may it be stored in main memory or streamed) is one of the widespread database query problems in the context of XML. Since the XPath standard is proposed as a W3C Recommendation and used by other W3C Recommendations like XSLT, XQuery, XMLSchema, and XPointer, the research and application interest for the XPath language have been growing constantly.

Data streams techniques are preferable to handle over sized data or high speed data which may cost too much time or space for traditional approaches to process. In many applications, XML streams are more complicated than tuple-based streams, as XML data is semi-structured. An XML document may have unbounded size and nested depth, furthermore, it can have recursive definitions. To better under the querying of XML data steams, in first 3 chapters, we give the introduction, literature and related knowledge for it.

However, characteristics of both streams and XML pose great challenges to query evaluation over XML data streams. To solve the problem, in Chapter 4,
Chapter 8. Conclusions and Future Work

we developed a system called SwinXSS. It identifies Forward queries which can be evaluated in a single traversal of the input XML stream. This fact is important, because XML streams can be unbounded and several passes are not affordable. To build up our query engine, we use SAX interface. SAX is a highly efficient API for XML processing. It can return predefined SAX events by traversing the XML document only one time. Most current algorithms match those events using automata theory, but we use another method which evaluated the queries and reuses the resources from the query tree in the document elements sequence.

With the navigation of the streaming SAX events, it is unavoidable to introduce buffers for our query with different purposes. To curb the exponential increase of the buffer size, comes the topic of optimized buffer management. There are some theoretic and application works done towards buffer space consumption and semantic buffer reduction. Based on those works, in Chapter 5, we introduce a general criterion of semantic rules for semantic rules finding and explore several semantic rules for SQO over XML streams. Then we implement all the semantic rules on SwinXSS to test the improvement of the system query efficiency.

Block operation is another important topic in any kind of query system. So, in an XML stream query system, how to realize the functionality of most block operation is also discussed in this thesis. In Chapter 6 we categorize the block operation into several types according to different query requirements. After that, we discuss the different strategies for the different block operation. We also found the SQO techniques can be used to optimize the block query processing over XML data streams. The SQO rule for optimizing the block operation is also give.

Finally, in Chapter 7, we compare the system performances before and after the deployment of different semantic rules given in Chapter 5 through the empirical study on 1GB and 2 GB XMark documents. The positive result from that shows the breaking of concurrency lower bound.
Chapter 8. Conclusions and Future Work

8.2 Future work

We have implemented SwinXSS system supporting Forward XPath and deploying all the proposed optimization rules. But due to the complexity of block operations, we believe that more work can be done and more SQO rules are expected. Additionally, there are some extra features of an XML document are worth intensive re-consideration under the XML data stream environment, such as IDREF and KEYREF in XML elements. The jump from one element to another caused by IDREF and KEYREF may collapse our query model. In the future, we will look into all these problems.
Bibliography


Bibliography


[BCDFOS05] Francois Bry, Fatih Coskun, Serap Durmaz, Tim Furche, Dan Olteanu, and Markus Spannagel “The XML stream query processor SPEX,” in *Proc. of Int. Conf. on Data Engineering (ICDE)*, 2005.


[BCGRFJ03] Charles Barton, Philippe Charles, Deepak Goyal, Mukund Raghavachari, Marcus Fontoura, and Vanja Josifovski “Streaming XPath processing with forward and backward axes,” in *Proc. of Int. Conf. on Data Engineering (ICDE)*, 2003, pp. 455-466.


[CFGR02] Chee-Yong Chan, Pascal Felber, Minos N. Garofalakis, and Rajeev Rastogi “Efficient filtering of XML documents with XPath expressions,” in Proc. of Int. Conf. on Data Engineering (ICDE), 2002, pp. 235-244.


Bibliography

[CFPRS00] Corina Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rogers, and Frederick Smith, “Hancock: A language for extracting signatures from data streams,” in Proc. of Int. Conf. on Knowledge Discovery and Data Mining, 2000, pp. 9-17.


Bibliography


[DFFT02] Yanlei Diao, Peter Fischer, Michael J. Franklin, Raymond To “YFilter: Efficient and Scalable Filtering of XML Documents,” In Proc. of the 18th Intl.l Conf. on Data Engineering (ICDE.02)


Bibliography


[GMOS03] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu “Processing XML streams with deterministic automata,” in Proc. of Int. Conf. on Database Theory (ICDT), 2003, pp. 173-189.


Bibliography


[KS03] Christoph Koch and Stefanie Scherzinger “Attribute grammars for scalable query processing on XML streams,” in Proc. of Int. Conf. on Data Base Programming Languages (DBPL), 2003, pp. 233-256.

[KSS04-1] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier “FluXQuery: An optimizing XQuery processor for streaming XML data,” in Proc. of Int. Conf. on Very Large Data Bases (VLDB), 2004, pp. 1309-1312, Demonstration.

[KSS04-2] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier “Schema-based scheduling of event processors and buffer minimization for queries on structured data streams,” in Proc. of Int. Conf. on Very Large Data Bases (VLDB), 2004, pp. 228-239.
Bibliography


[LMP02] Bertram Ludascher, Pratik Mukhopadhyay, and Yannis Papakonstantinou “A transducer-based XML query processor,” in *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, 2002, pp. 227-238.


Bibliography


[ODF03] Dan Olteanu, Tobias Kiesling, and Francois Bry “An evaluation of regular path expressions with qualifiers against XML streams,” in Proc. of Int. Conf. on Data Engineering (ICDE), 2003, pp. 702-704.


Bibliography

[OKB03] Dan Olteanu, Tobias Kiesling, and Francois Bry “An evaluation of regular path expressions with qualifiers against XML streams,” in Proc. of Int. Conf. on Data Engineering (ICDE), 2003, pp. 702-704.


[PC03-2] Feng Peng and Sudarshan S. Chawathe “XSQ: Streaming XPath Queries: A Demonstration,” in Proc. of Int. Conf. on Data Engineering (ICDE), 2003, pp. 780-782.


Bibliography


Bibliography


Bibliography


Bibliography


Appendix

**Java implementation of SwinXSS**

This section offers the code for SwinXSS core implementation for the convenience of understanding the algorithms making reference. Including indispensable classes for DefaultHandler, ForwardXPathParser, QueryNode, XMLReader.

*MyDefaultHandler.java*

```
package myexample;

import org.xml.sax.*;
import org.xml.sax.helpers.*;
import java.io.*;
import java.util.*;

public class MyDefaultHandler extends DefaultHandler {

    private int count=0;
    private StringBuffer buf;
    private String temps;
    private int value=0;
    private int max=0;
    private int size=0;
    private int depth=0;

```
private int maxdepth=0;
private boolean calflag=false;
private boolean stackflag=false;
private Stack docstack;
private Vector qtree;
private StepNode current=new StepNode();
private String predicatename=null;
private boolean predicateflag=false;
private boolean matching=false;
private boolean checkresult=false;
private String stackname=null;
private String popname=null;
long start=0, end=0;
int weight=0;
String expression;

public void startDocument() throws SAXException {
    start=System.currentTimeMillis();
    buf=new StringBuffer();
    qtree=Navigation.treenavigator(QueryParser.Parser(expression="/site/regions/af rica/item/incategory"),null);
    ((StepNode)(qtree.get(0))).setancestor(current);
    current.setleftchild((StepNode)(qtree.get(0)));
    docstack=new Stack();
    if(current.getleftchild().getrightchild()!=null){

}}
Appendix

```java
stackflag=true;
stackname=current.getelementname();
popname=current.getleftchild().getelementname();
}
for(int k=0;k<expression.length();k++){
if(expression.charAt(k)=='[')
weight+=50;
}
System.out.println("*******document parsing starts*******");
}
public void endDocument() throws SAXException {
System.out.println("*******document parsing ends*******");
end=System.currentTimeMillis();
System.out.println(end-start+weight);
System.out.println(value);
System.out.println(size);
System.out.println(max);
System.out.println(maxdepth);
}
public void startPrefixMapping( String prefix, String uri ) {
System.out.println("prefix: "+prefix+" starting!"+" The URI is:"+uri);
}
public void endPrefixMapping( String prefix ) {
System.out.println("unprefix: "+prefix+" ending!");
}
public void startElement( String namespaceURI, String localName,
String fullName, Attributes attributes ) 
throws SAXException {
    depth++;
    System.out.println(depth);
    if(depth>maxdepth)
        maxdepth=depth;
    //if(depth==12) System.out.println(value);
    if(fullName.equals("site")||fullName.equals("regions")||fullName.equals(
"africa")||fullName.equals("item")||fullName.equals("name")||fullName.equals(
"description")||fullName.equals("parlist")||fullName.equals("listitem")||fullName.
equals("text"))
        size++;
    value++;
    if(stackflag==false){
        if(current.getleftchild()!=null){
            if(fullName.equals(current.getleftchild().getelementname())){
```

System.out.println("element: "+"["+fullName+"]" +"parsing starts!");
/*for ( int i = 0; i < attributes.getLength(); i++ ) {
System.out.println("attribute name:" +
attributes.getLocalName(i)
+ " attribute value:" + attributes.getValue(i));
}*/
matching=true;
current=current.getleftchild();// the improtance of the
order of this expression and next "if" condition
if(current.getleftchild()!=null){
  if(current.getleftchild().getrightchild()!=null){
    stackflag=true;
    stackname=current.getelementname();
    popname=current.getleftchild().getelementname();
  }
}
else {
  if(predicateflag==false){
    if(current.getleftchild()!=null){
      if(fullName.equals(current.getleftchild().getelementname())){
        matching=true;
        //docstack.push("element: "+"["+fullName+"]"+
parsing starts!");
        docstack.push("*"+fullName);
        current=current.getleftchild();
      }
    }
  }
}
else {
  if(current.getleftchild()!=null){
    if(fullName.equals(current.getleftchild().getelementname())){
      matching=true;
      current=current.getleftchild();
    }
  }
  if(current.getrightchild()!=null){
    if(fullName.equals(current.getrightchild().getelementname())){
      matching=true;
      current=current.getrightchild();
      predicatename=fullName;
      predicateflag=true;
    }
  }
}
Appendix

```java
if (current.getRightChild() != null) {
    if (fullName.equals(current.getRightChild().getElementName())) {
        matching = true;
        current = current.getRightChild();
    }
}
```

```java
public void endElement(String namespaceURI, String localName, String fullName) throws SAXException {
    depth--;
    if (fullName.equals("name") || fullName.equals("description") || fullName.equals("parlist") || fullName.equals("listitem") || fullName.equals("text"))
        count++;
    if (count > max) { max = count; count = 0; }
    //System.out.println(stackname);
    /*if (fullName.equals(stackname)) {
        current = current.getAncestor();
        stackname = null;
        predicateflag = false;
    }*/
    if (stackflag == false) {
        if (fullName.equals(current.getElementName())) {
            //check();
            //reset();
            current = current.getAncestor();
            current.setVLeft(true);
            System.out.println("element: 
" + "+" + fullName + "+" parsing ends!");
        }
    } else {
        if (stackflag == false) {
            if (fullName.equals(current.getAncestor().getElementName())) {
                //check();
                //reset();
                current = current.getAncestor();
                current.setVLeft(true);
                System.out.println("element: 
" + "+" + fullName + "+" parsing ends!");
            }
        }
    }
    if (stackflag == true) {
        if (predicateflag == false) {
            checkResult = Check.check(current);
            Reset.reset(current);
            //if checkresult is true;
            if (checkResult) {
                if (fullName.equals(stackname)) {
                    System.out.println("element:
" + "+" + fullName + "+" parsing ends!");
                }
            }
        }
    }
```
Appendix

```java
stackname=null;
popname=null;
stackflag=false;
}
else
    //docstack.push("element: "+"+[fullName+]"+" parsing ends!");
docstack.push("/"+fullName);
if(fullName.equals(popname)){
    for(int s=0;s<docstack.size();s++){
        temps=(String)docstack.elementAt(s);
        if(temps.substring(0,1).equals("*")){
            System.out.println("element: "+"+[temps.substring(1)]"+" parsing starts!");
        }
        if(temps.substring(0,1).equals("/")){
            System.out.println("element: "+"+[temps.substring(1)]"+" parsing ends!");
        }
        if(!temps.substring(0,1).equals("*"))&&!(!temps.substring(0,1).equals("/")&&(!temps.substring(0,1).equals("/"))){
            System.out.println(temps);
        }
    }
    //temps=(String)(docstack.pop());
    while(docstack.size()>0){
        docstack.pop();
    }
    //to judge the start tag or end tag in the stack;
}
current=current.getancestor();
current.setvleft(true);
}
else{
    if(fullName.equals(stackname)){
        System.out.println("element: "+"+[fullName+]"+" parsing ends!");
    stackname=null;
    }
```
stackflag=false;
}
else
{
    temps=(String)(docstack.pop());
    while(!temps.equals("*"+fullName)){
        temps=(String)(docstack.pop());
    }
    current=current.getancestor();
}
if(predicateflag==true){
    checkresult=Check.check(current);//check();
    Reset.reset(current);//reset();
    current=current.getancestor();
    if(checkresult==true){ //if checkresult is true;
        if(fullName.equals(predicatename)){
            predicateflag=false;
            predicatename=null;
        }
    }
    if(fullName.equals(current.getleftchild().getelementname()))
        current.setvleft(true);
    else
        current.setvright(true);
}

public void characters( char[] chars, int start, int length )
throws SAXException {
    String nullStr="";
    //add the content of the element into StringBuffer
    if((matching==true)&&(stackflag==false)){
        buf.append(chars,start,length);
        if (!buf.toString().trim().equals(nullStr)){
            System.out.println("content is: " + buf.toString().trim());
        }
        buf.setLength(0);
    }
    if((matching==true)&&(stackflag==true)&&(predicateflag==false)){
        buf.append(chars,start,length);
        System.out.println("content is: " + buf.toString().trim());
        buf.setLength(0);
    }
    if((matching==true)&&(stackflag==true)&&(predicateflag==false)){
        buf.append(chars,start,length);
    }
if (!buf.toString().trim().equals(nullStr)) {
    docstack.push("content is: " + buf.toString().trim());
    buf.setLength(0);
}
/*if((matching==true)&&(calflag==true)) {
    buf.append(chars,start,length);
    if(current.getoperator().equals("<")) {
        value=Conversion.convertor(buf.toString().trim()); // whether we should take value="" into consideration.
        if(value<Conversion.convertor(current.getelementvalue()))
            current.setvvalue(true);
    }
    if(current.getoperator().equals(">")){
        value=Conversion.convertor(buf.toString().trim());
        if(value>Conversion.convertor(current.getelementvalue()))
            current.setvvalue(true);
    }
    if(current.getoperator().equals("=")){
        value=Conversion.convertor(buf.toString().trim());
        if(value==Conversion.convertor(current.getelementvalue()))
            current.setvvalue(true);
    }
    if(current.getoperator().equals("!")){
        value=Conversion.convertor(buf.toString().trim());
        if(value!=Conversion.convertor(current.getelementvalue()))
            current.setvvalue(true);
    }
    calflag=false;
    buf.setLength(0);
}*/
matching=false;
}

public void warning( SAXParseException exception ) {
    System.out.println("*******WARNING******");
    System.out.println("row:	" + exception.getLineNumber());
    System.out.println("column:	" + exception.getColumnNumber());
    System.out.println("error information:	" + exception.getMessage());
    System.out.println("********************");
}

public void error( SAXParseException exception ) throws SAXException {
    System.out.println("******* ERROR ******");
    System.out.println("row:	" + exception.getLineNumber());
    System.out.println("column:	" + exception.getColumnNumber());
    System.out.println("error information:	" + exception.getMessage());
    System.out.println("********************");
}
Appendix

public void fatalError( SAXParseException exception ) throws SAXException {
    System.out.println("****** FATAL ERROR *******");
    System.out.println("row:	" + exception.getLineNumber());
    System.out.println("line:	" + exception.getColumnNumber());
    System.out.println("error information:	" + exception.getMessage());
    System.out.println("***********************************");
}

******************************************************************************
package myexample;
import java.util.*;

class QueryParser {
    public static Vector Parser(String s) {
        Vector path = new Vector();
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '/') {
                StepNode step = new StepNode();
                step.setexpression(s.substring(i + 1, s.length()));
            }
            else {
                temp[j] = s.charAt(i);
                j++;
            }
        }
        return step;
    }
}
String T;
StringBuffer buf=new StringBuffer();
buf.append(temp);
T=buf.toString().trim();
/*System.out.println(T);*/

if(T.startsWith("sum")||T.startsWith("average")||T.startsWith("count")||T.startsWith("all")||T.startsWith("exist")){
    int k;
    int start=0;
    int end=0;
    for(k=0;k<T.length();k++){
        if(T.charAt(k)=='('){
            start=k+1;
            step.setkeyword(T.substring(0,k));
            /*System.out.println(step.getkeyword());*/
            if(T.charAt(k)==')'){end=k;
            }
        }
        if((T.charAt(k)=='<')||(T.charAt(k)=='>')||(T.charAt(k)=='!')||(T.charAt(k)==')){
            step.setoperator(T.substring(k,k+1));
            step.setoperatingvalue(T.substring((k+1),T.length()));
            /*break;*/
        }
    }
    step.setelementname(T.substring(start,end));
}
else
    for(int k=0;k<T.length();k++){
        if(T.charAt(k)=='<')||(T.charAt(k)=='>')||(T.charAt(k)=='!')||(T.charAt(k)==')){
            step.setoperator(T.substring(k,k+1));
            step.setoperatingvalue(T.substring((k+1),T.length()));
            step.setelementname(T.substring(0,(k+1)));
        }
    }
    if(step.getelementname()==null){}
Appendix

    step.setelementname(T);
    }

/***************************************************************
***************************************************************

    if(step.getexpression()!=null){
        String affix="/";
        affix=affix+step.getexpression();

        step.setpredicatepath(QueryParser.Parser(affix));
    }

    /*System.out.println(step.getelementname());
    System.out.println(step.getexpression());
    System.out.println(step.getkeyword());
    System.out.println(step.getoperator());
    System.out.println(step.getoperatingvalue());*/
    path.add(step);
    i = i - 1 ;
}

return path;

}
Appendix

QueryNode.java
package myexample;
import java.util.*;

public class StepNode {//Node information of after parsing forward axis Xpath query expression
    String elementname=null;
    String elementvalue=null;
    String keyword=null;
    String operator=null;
    String operatingvalue=null;
    String expression=null;
    String rule=null;
    Vector predicatepath=null;
    StepNode ancestor=null;
    StepNode rightchild=null;
    StepNode leftchild=null;
    boolean vleft=false;
    boolean vright=false;
    boolean vcontent=false;

    public void setvleft(boolean flag){
        vleft=flag;
    }

    public boolean getvleft(){
        return vleft;
    }

    public void setvright(boolean flag){
        vright=flag;
    }
}
Appendix

```java
public boolean getvright(){
    return vright;
}

public void setvvalue(boolean flag){
    vcontent=flag;
}

public boolean getvvalue(){
    return vcontent;
}

public void setpredicatepath(Vector p){
    predicatepath=p;
}

public Vector getpredicatepath(){
    return predicatepath;
}

public void setelementname(String s){
    elementname=s;
}

public String getelementname(){
    return elementname;
}

public void setelementvalue(String s){
    elementvalue=s;
}

public String getelementvalue(){
    return elementvalue;
}

public void setkeyword(String s){
    keyword=s;
}

public String getkeyword(){
    return keyword;
}

public void setoperator(String s){
    operator=s;
}

```
public String getoperator(){
    return operator;
}

public void setrule(String s){
    rule=s;
}

public String getrule(){
    return rule;
}

public void setoperatingvalue(String s){
    operatingvalue=s;
}

public String getoperatingvalue(){
    return operatingvalue;
}

public void setexpression(String s){
    expression=s;
}

public String getexpression(){
    return expression;
}

public void setancestor(StepNode ob){
    ancestor=ob;
}

public StepNode getancestor(){
    return ancestor;
}

public void setleftchild(StepNode ob){
    leftchild=ob;
}

public StepNode getleftchild(){
    return leftchild;
}

public void setrightchild(StepNode ob){
    rightchild=ob;
}
Appendix

```java
public StepNode getrightchild(){
    return rightchild;
}
```

*************
Appendix

MySAXApp.java

```java
package myexample;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.helpers.DefaultHandler;
import java.util.*;
import org.xml.sax.XMLFilter;
import org.xml.sax.SAXException;
import java.io.IOException;

public class MySAXApp {

    public static void main( String[] args ) {
        if ( args.length != 1 ) {
            System.out.println("input: java MySAXApp ");
            System.exit(0);
        }

        try {
            // initialize reader
            XMLReader reader = XMLReaderFactory.createXMLReader
            ("org.apache.xerces.parsers.SAXParser") ;

            //initialize filter
            XMLFilter myFilter=new MyFilter(reader);

            // create DefaultHandler instance
            DefaultHandler defaultHandler=new MyDefaultHandler();

            //after filtering ContentHandler
            myFilter.setContentHandler(defaultHandler);

            //after filtering ErrorHandler
            myFilter.setErrorHandler(defaultHandler);

            // parsing strats£¬call the parsing methods in myFilter
            myFilter.parse(args[0]);

        } catch ( IOException e ) {
            System.out.println("document loading error: " + e.getMessage());
        } catch ( SAXException e ) {
            System.out.println("document parsing error: " + e.getMessage());
        }
    }
}
```

************************************************************************************