University of California
Los Angeles

Efficient Support for XML Queries on Data Base and Data Stream Management Systems

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Xin Zhou

2006

The dissertation of Xin Zhou is approved.

Alfonso F. Cardenas

Junghoo (John) Cho

Anne J. Gilliland

Carlo Zaniolo, Committee Chair

University of California, Los Angeles

2006

Table of Contents

List of Figures

List of Tables

Vita

1976	Born, Suzhou, Jiangsu, China
1998	B.S. in Computer Science Fudan University, Shanghai, China
2001	M.S. in Computer Science Fudan University, Shanghai, China
2002	M.S. in Computer Science University of California, Los Angeles, USA
2006	PH.D. in Computer Science University of California, Los Angeles, USA

Publications

Xin Zhou, Fusheng Wang, Carlo Zaniolo, *Efficient Temporal Coalescing Query Support in Relational Database Systems*. In Proc. of DEXA 2006.

Xin Zhou, Hetal Thakkar, Carlo Zaniolo, *Integrated Processing of XML and Relational Data Streams*. Poster Paper. In UCLA Engineering Research Review, Los Angeles, USA, May 2006.

Xin Zhou, Hetal Thakkar, Carlo Zaniolo, *Unifying the Processing of XML Streams and Relational Data Streams*. In Proc. of ICDE 2006.

Fusheng Wang, Xin Zhou and Carlo Zaniolo, *Using XML to Build Efficient Transaction-Time Temporal Database Systems on Relational Databases*. Poster Paper. In Proc. of ICDE 2006.

Fusheng Wang, Xin Zhou and Carlo Zaniolo, *Efficient XML-based Techniques for Archiving Querying and Publishing the History of Relational Databases*. Time-Center Technique Report, TR-83, June 2005.

Fusheng Wang, Carlo Zaniolo, Xin Zhou, Hyun Jin Moon, *Managing Multi-Version Documents Historical Databases: a Unified Solution Based on XML*. Demo Paper. In Proc. of WebDB 2005.

Fusheng Wang, Xin Zhou and Carlo Zaniolo, *Using XML to Build Efficient Transaction-Time Temporal Database Systems on Relational Databases*. Time-Center Technique Report, TR-81, March 2005.

Fusheng Wang, Carlo Zaniolo, and Xin Zhou, *Temporal XML? SQL Is Fighting Back!*. In Proc. of TIME 2005.

Fusheng Wang, Carlo Zaniolo, Xin Zhou, Hyun Jin Moon, *Version Management and Historical Queries in Digital Libraries*. Poster Paper. In Proc. of TIME 2005.

Fusheng Wang, Xin Zhou and Carlo Zaniolo, *Temporal Information Management using XML*. Demo Paper. In Proc. of ER 2004.

Abstract of the Dissertation

Efficient Support for XML Queries on Data Base and Data Stream Management Systems

by

Xin Zhou

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2006

Professor Carlo Zaniolo, Chair

The eXtensible Markup Language (XML) is rapidly becoming the de facto standard for retrieving and exchanging web information, and is having a significant impact on the evolution of existing database management systems. Commercial SQL-compliant data base systems are moving aggressively to support the integrated management of XML documents and XML-published relational data. This is achieved by supporting XML query languages, such as XQuery, along with extensions of relational languages such as SQL/XML.

In this dissertation, we focus our work on three areas that present research issues of particular significance and interest. The first area involves managing and querying the temporal history of databases using XML, since its data model and query languages are more supportive of temporal information than relational databases and SQL. We then discuss the problem of supporting this XML historical view efficiently by mapping back to relational databases, since these achieve better scalability performance. We finally introduce an algorithm which can

process temporal coalescing efficiently, within the current SQL framework.

The second and fast growing opportunity area is represented by data streams, whereby query languages need to be extended to support continuous query applications on incoming relational streams and XML streams. While current approaches design separate data stream management systems for relational and XML data, the focus of my research has been on providing a unified support for both kinds of data streams. Toward this goal, XML SAX event streams and relational streams need to be integrated, and SQL and XQuery must be extended to overcome their limitations with certain data stream queries. We also overview the state-of-the-art technologies based on Finite State Automata (FSA) model to process multiple queries, and use UDAs to simulate such FSAs with comparable performance. These techniques allow us to build a system which integrated the management of relational and XML streams.

The last but not the least research problem in my thesis is OLAP applications on XML data. With XML gaining importance as the standard for representing business data, XQuery must support the types of queries that are common in business analytics. One such class of queries is OLAP-style aggregation queries. We review several recent research proposals which introduce new constructs into XQuery, pinpoint their disadvantages, and bring forward our function import mechanism. Basically, we allow XQuery UDFs to import functions written in SQL:2003 and UDAs. All complex moving window queries and grouping set queries can thus be specified in SQL easily using current SQL standards, and optimize using the mature query optimization technology of relational DBMSs. We show our approach provides a clear language expressive power and efficient query performance.

CHAPTER 1

Introduction

1.1 Overview

The eXtensible Markup Language (XML) is a standard for data exchange, which in recent years, has attracted significant interest from industrial and research forums. Applications increasingly rely on XML not only to exchange but also to query, filter and transform data [42]. Thanks to standard specifications for web services (such as SOAP [11], WSDL, etc.) applications can receive requests for data (specified in XML) and return their answers tagged as XML documents. In addition, by exploiting XML query languages such as XPath [10] and XQuery [13], users and applications can compose declarative specifications of their interests as well as filter and transform specific data items represented in XML.

The reasons behind XMLs popularity are not surprising. The ability to exchange data between applications, systems and other entities has always been of vast importance. Before XML became a reality, the need for data exchange was met with proprietary techniques defined and supported by specific vendors. The emergence of XML as a standard and the development of a core set of specifications around XML enabled the steady transition from proprietary data representations and languages to XML-enabled suites of applications. XML is flexible, extensible, and self-describing; and as such, it is suitable for encoding data in a format (including both structure and content types) that senders and re-

ceivers can agree upon. Such XML-based external formats enable heterogeneous systems to exchange data without knowing how the data is internally represented in the individual systems. As a result, XML has emerged in the industry as the predominant mechanism for representing structured and semi-structured information for exchanging between applications across the Internet, and within an intranet. Virtually every industry is working to standardize XML representations for their common business objects.

XQuery [13] and SQL/XML [8] are the two industry-standard languages that have emerged to retrieve and query business documents encoded in XML. XQuery provides a rich query language that supports the hierarchical structure of XML. SQL/XML extends the relational model with XML data type constructs to query relational data, and to convert between relational and XML data. Despite the slightly different focus of the two languages, they both include many similar concepts including set-based and sequence- based processing, joins, selections, projections, and quantification.

A large percentage of industries rely heavily on existing relational databases and applications to run their businesses, from which much of the information within the XML document is generated, or into which much of the information from the XML documents will be stored [20]. The integration of this well-structured relational information with the self-describing XML data represent important evolutionary trend in the data processing and warehousing industry. It is widely accepted that XML and relational data will coexist and complement each other in supporting enterprise solutions.

Another vibrant area of database research [49] is represented by data streams. The need to support applications that span traditional databases and data streams provides the rationale for the approach taken by most projects of extending query

languages and data models to support efficiently continuous queries on stream data. Inasmuch as the integration of XML and its query languages into traditional DBMSs represent a fundamental trend for commercial DBMS and database research [20], we expect that similar integration of XML streams and relational data streams would be one of the critical research area for future data intensive applications.

1.2 Challenges

From the many widespread XML application areas, we focus on three which are of particular importance and technical interests. These are: (i) XML application in temporal database management, (ii) integration of XML and relational streams, and (iii) OLAP extension for XML. In the course of our research work, we encountered several challenges which are listed as follows:

- Efficient storage model and query translation for historical XML document:

  We consider two alternative solutions for supporting temporal database applications. One is to use a native XML database system to store and query efficiently a large-size historical XML document. The second is to shred the XML document into relational tables. With the second approach, we need to address a lot of issues related to find the balance between logical representation and physical implementation. Specifically, based on different table storage models, we need to develop the optimal translation engine for all kinds of temporal queries. To make our temporal database model convincing enough, we also need to solve the classical temporal coalescing problem in our framework.

- Integrate the processing for XML and relational streams:

Relational data streams and XML streams have previously provided two separate research foci, but their unified support by a single Data Stream Management System (DSMS) is very desirable from an application viewpoint. To support such an integration, we need to handle two natively different data models, one based on flat tables, and the other based on hierarchies. Even if we have an integrated data stream, how can we obtain an integrated query language is much more difficult - and the problem becomes even worse because of XQuery's limited expressive power in data stream applications. In addition, how to cooperate in one system the current techniques for efficient multi-query processing represents another challenging issue.

- An effective OLAP extension in XML applications:

  With the wide acceptance of XML as the standard business information exchange format, with XQuery as its main query language, it is very desirable to support OLAP functions in XQuery. However, due to its lack of explicit grouping functionality, its navigation-oriented FLOWR structure, and the fact that the function definition mechanism is blocking - XQuery is not well suited for OLAP queries and continuous queries on moving windows. To solve this problem, there have been recent proposals to introduce constructs such as GROUP BY and NEST in XQuery [57]. However, the proposed constructs require significant changes to current XML standards, and are not as easy to use or to optimize as their relational counterparts. As a result, we need to come up with a new solution to this problem, taking both expressiveness and efficiency into consideration.

1.3 Main Contributions

This dissertation addresses the three technical challenges outlined above as follows:

- A unified multi-model support for historical information management.

  Our unified efficient implementation for historical information management using relational models relies on advanced query mapping/optimization techniques, and temporal clustering/indexing techniques at the internal level. These techniques have been tested and optimized by developing ArchIS system [89, 85], which provides an efficient approach to address a wide range of applications. The system also contains the flexibility of three different temporal views, as follows: (1) XML-based views which dovetail with web applications, while (2) nested-relations are more natural for object-oriented applications, and (3) null-filled flat tables are best for traditional database applications, decision support applications, and event-oriented queries. This last approach provides a simple framework for the presentation of the data, which can require significantly more effort when XML is used. To support classical temporal coalescing queries, we develop a SQL:2003-based query which only requires one single scan of the input data tuples. We discuss all of these issues in Chapter 3.

- A UDA-based approach for stream integration and multi-query processing.

  We use relational user-defined aggregates (UDA) [92, 61], a native SQL compatible aggregation definition language, to implement a unified data stream system and support multi-query processing. Specifically, in our Stream Mill system [28, 94], XML streams expressed as SAX events, can be

flattened into relational streams, and vice versa, by using UDAs. This enables a close cooperation of their query languages, resulting in great power and flexibility. Many benefits are also gained at the system level, since relational DSMS techniques for load shedding, memory management, query scheduling, approximate query answering, and synopsis maintenance can now be applied to XML streams. Moreover, the many FSA-based optimization techniques developed for XPath and XQuery can be easily and efficiently incorporated in our system. Indeed, we show that YFilter, which is capable of efficiently processing multiple complex XML queries, can be easily integrated in Stream Mill via ESL user-defined and system-defined aggregates. This approach produces a powerful and flexible system where relational and XML streams are unified and processed efficiently. We summarize our work in this area in Chapter 4.

- A function import mechanism to enable XQuery with OLAP functionalities.

  To solve the problem of XQuery's limitation in terms of expressive power for grouping queries, we propose a simpler approach that allows XQuery user-defined functions to import OLAP constructs via SQL/XML [93]. This approach achieves great power and flexibility without requiring significant extensions or new constructs in XQuery. Indeed, in Chapter 5, we show that along with value-based OLAP queries that are similar to those of relational database, we also support OLAP queries that exploit the unique structure of XML. This solution also overcomes the blocking problem of XQuery aggregate functions, enhancing its ability to support continuous queries such as moving windows on XML documents. Thus we can support SQL:2003 [15, 7] analytic functions using the logical/physical window constructs that have proved so useful in both relational applications and data

stream applications.

1.4 Dissertation Organization

This dissertation is organized as follows. After the brief introduction in this chapter, Chapter 2 reviews the state of the art, and pinpoints three interesting areas where we develop our research work. Chapter 3 summarizes our work in managing historical information using XML and relational databases. We focus on the relational storage model of a historical XML view, temporal query translation, and the efficient processing of typical temporal coalescing queries using SQL queries. In Chapter 4, we discuss our work in designing and developing a system that unifies the processing for relational and XML data stream, including data unification, query language integration, and efficient query support for multi-query processing. OLAP support in XML is another interesting problem, and we delve into this problem in Chapter 5. After a critique of recent proposals to introduce OLAP extensions to XML, we describe how moving window queries and grouping set queries can be easily supported in XQuery by taking advantage of SQL/XML standards. Chapter 6 concludes our dissertation.

CHAPTER 2

State of the Art

In this chapter, we will provide the technical context for the work presented in the subsequent chapters. The basic data model and query languages of XML are briefly summarized in Section 2.1. A lot of interesting research problems remain open in XML related topics, including XML data storage model, query processing, integration with relational techniques, and various database and business applications. Among these wide assortment of research problems, we devote our research work to three areas of special technical interest and practical significance. In Section 2.2, we review the problem of version management and temporal queries. We will see in later chapters how we tackle this problem with XML representation and efficient query processing for temporal applications. In Section 2.3, we study the important problem of streaming XML processing and data stream management systems, and review its current research status. Complex OLAP grouping queries have recently become very hot in both academic areas and standardization organizations, and we discuss this problem in Section 2.4.

2.1 XML Data Models and Query Languages

2.1.1 XML Data Model

The eXtensible Markup Language, abbreviated as XML, is a self-describing, flexible, and extensible text format that was originally designed to meet the challenges of large-scale electronic publishing. XML is playing an increasingly important role in the exchange of a wide variety of data on the Internet and in enterprise intranets.

XML describes a class of data objects that are called XML documents. XML messages can be viewed as a sequence of XML documents. XML provides a mechanism for tagging document content to provide a detailed description of its organization. Specifically, XML allows a document to take a hierarchical structure that consists of a root element and sub-elements; elements can be nested to any depth. Figure 2.1 shows an example XML document that contains books information. In this example, the root element is *book*; it contains sub-elements such as *title*, *author*, *publisher*, and *category*. *category* in turn contains other sub-elements under it.

An XML element starts with a start-tag enclosed by a pair of angle brackets. The start tag consists of a tag name and an optional list of attribute specifications. In Figure 2.1, the *book* element contains an attribute *id* with a value "DC1001". An XML element ends with a matching end-tag that is marked by a "/" symbol before the tag name in its enclosing brackets. The content of an element resides between its start and end tags, and can contain not only sub-elements but also text data. For example, the title element in Figure 2.1 has the text data "A Complete Guide to DB2 Universal Database".

A general set of rules for a documents elements and attributes can be defined

```
<book id = "DC1001">
   <title>A Complete Guide to DB2 Universal Database</title>
   <author>Don Chamberlin</author>
   <publisher>Morgan Kaufmann Publishers, Inc</publisher>
   <year>1999</year>
   <category>
      <software>
        <db><concurrency/></db>
      </software>
      <distributed/>
   </category>
   <price>$70.00</price>
</book>
```

Figure 2.1: book.xml: A sample XML document containing book information

in a Document Type Definition (DTD) [52] or an XML schema [12]. A DTD specifies information about a class of documents including all possible structures; in addition to this, XML schema allows to support the domains of values that attributes in those documents can take.

XML's tagging mechanism and associated technologies for defining rules for such tagging result in three key properties of XML: self-description, flexibility, and extensibility. XML is self-describing because it supports the use of element tags to describe document content. XML is flexible because DTDs and XML schemas allow significant variance in the structure and content of documents; for example, an element, attributes of an element, or text data of an element can be optional, and elements of the same tag name can appear multiple times inside the same enclosing element. Furthermore, XML is extensible because DTDs and XML schemas can be defined and modified by any user. This extensibility is fundamentally different from the HyperText Markup Language (HTML), which

uses a pre-defined, fixed set of tags. It is these three properties that have pushed XML to the forefront of electronic publishing and online information exchange.

2.1.2 XML Query Language

Having described XML document structure, we now present the standard XML query language.

XQuery [13] is a declarative language for querying XML data. It is designed to be broadly applicable to many types of XML data sources. XQuery is commonly used to locate and extract elements and attributes from XML documents and also to construct new XML documents using the extracted entities. It is expected that update functionality will be included into XQuery language soon.

2.1.2.1 Path Expressions

A basic, common form of XQuery expressions are path expressions that can contain constraints over both structure and content of XML fragments, which are called XPath [10]. Path expressions are based on a view in which an XML document is a tree of nodes. Given this view, path expressions are essentially patterns that are matched to nodes in the XML tree. A path expression consists of a sequence of one or more location steps. Each location step consists of an axis, a node test and zero or more predicates. An axis specifies the hierarchical relationship between nodes. This dissertation focuses on two common axes: the child axis / (i.e., nodes at adjacent levels), and the descendent axis // (i.e., nodes separated by any number of levels). In the simplest and most common form, a node test is a name test, which is specified by either an element name or a wildcard operator (*) that matches any element name.

Each location step can also include one or more predicates to further refine the selected set of element nodes. A predicate, delimited by "[" and "]" symbols, is applied to the element node addressed at a location step. Predicates can specify constraints on the text data or the attributes of the addressed element nodes. In this dissertation, such predicates are referred to as value-based. In addition, predicates may also include other path expressions, which are called nested path expressions. Nested paths are relative paths with respect to the location steps where their enclosing predicates reside; accordingly, they are evaluated in the context of each of the element nodes that their enclosing predicates address.

For a simple example, consider a user who is interested in the title of a book. Example 2.1 below (based on the schema in Figure 2.1) expresses such a query. This query specifies that the root element of the document must be *book* and returns all the *title* sub-elements under *book*.

Example 2.1 *Return the title information of every book (XPath).*

/book/title

Based on the data in Figure 2.1, the answer returned by the query of Example 2.1 will be:

<title>A Complete Guide to DB2 Universal Database </title>

Example 2.2 shows a much more complex query. Basically, the user wants some category information of books whose author is "Don Chamberlin". Predicate [$author$ = "$DanChamerlin$"] filters such request on the input elements. Specifically, the user wants every category description under the *category* element, so "//*" means every descendent elements under *category* should be returned.

Example 2.2 *Return the category information of books written by Don Chamberlin (XPath).*

*//book[author = "Dan Chamberlin"]/category//**

If multiple elements match a path query, the return will be a list of elements. As a result, the return of Example 2.2 based on data in Figure 2.1 is:

<software><db><concurrency/></db></software>

<db><concurrency/></db>

<concurrency/>

<distributed/>

2.1.2.2 FLWOR Expression

XQuery also allows customized XML documents to be created using For-Let-Where-Order By-Return expressions. Such FLWOR expressions are a high-level language construct that combines matching and restructuring of XML data. These expressions provide a powerful way to specify requirements for transforming XML messages.

- The *For* clause contains a variable name and a path expression;
- The *Let* clause contains a variable name and a path expression;
- The *Where* clause contains a set of conjunctive predicates, each of which takes the form of a triplet: a relative path expression, an operator, and a constant;
- The *Order By* clause contains a set of variables or paths; where all constant tags have a matching close tag.

The semantics of the FLWOR expression is as follows. The *For* clause creates an ordered sequence of variable bindings to element nodes. The *Let* clause further binds a list of elements satisfying its path expression to some variables. The *Where* clause, if present, restricts the set of bindings passed to the return clause. The *Order By* clause orders the iteration in the *For* clause by the value in the specified variables and paths. The *Return* clause is invoked once for each variable binding. At each invocation of the return clause, tags cause the construction of new element nodes and path expressions select nodes from the current variable binding; if multiple nodes are selected for a path expression, they are grouped and listed in their document order. The final result of the FLWOR expression is an ordered sequence of the results of these invocations.

Continuing on the previous examples, Example 2.2 can be written with the following equivalent FLWOR XQuery expression:

Example 2.3 *Return the category information of books written by Don Chamberlin (XQuery)*

FOR *$a* in *(doc("books.xml")/book)*
LET *$b := $a/author*
LET *$c := $a/category//**
WHERE *$b = "Dan Chamberlin"*
RETURN *<category>{$c}</category>*

The only difference between Example 2.2 and Example 2.3 is that in Example 2.3, for every element in the returned element list, a new tag *category* will be encapsulated.

A very important feature of XQuery is that it allows a lot of system defined functions such as *subsequence ()*, *count ()*, and native user-defined functions. We will discuss this feature in more detail in the following chapters.

After this brief review of the basic data model and query language for XML, we will now overview three interesting application areas, provide the main research focus of this dissertation.

2.2 Version Management and Temporal Queries

Our first application of XML data model and query language is in the area of temporal database management systems. This problem relates to the long-studied problems of version management and temporal databases. In this section, we will review version management and temporal queries.

2.2.1 Version Management

The management of multiple versions of documents finds important applications and poses interesting technical challenges. Indeed, important application domains, such as software configuration and cooperative work, have traditionally relied on version management. As these applications migrate to a web-based environment, they are increasingly using XML for representing and exchanging information, and often seeking standard vendor-supported tools and environments for processing and exchanging their XML documents.

Many new applications of versioning are also emerging because of the web; a particularly important and pervasive one is assuring link permanence for web documents. Any URL becoming invalid causes serious problems for all documents referring to it. Thus the problem is particularly severe for search engines that risk directing millions of users to pages that no longer exist. Replacing the old version with a new one, at the same location, does not cure the problem completely, since the new version might no longer contain the keywords used in the search. The

ideal solution is a version management system supporting multiple versions of the same document, while avoiding duplicate storage of their shared segments. For this reason, professionally managed sites and content providers will have to use document versioning systems; frequently, web service providers will also support searches and queries on their repositories of multiversion documents. Specialty warehouses and archives that monitor and collect content from websites of interest will also rely on versioning to preserve information, track the history of downloaded documents, and support queries on these documents and their history [43].

Various techniques for versioning have also been proposed by database researchers who have focused on problems such as transaction-time management of temporal databases [67], support for versions of CAD artifacts in object-oriented databases [59] and change management for semistructured information [36].

2.2.2 Temporal Data Model and Queries

There is a large number of temporal data models proposed and the design space for the relational data model has been exhaustively explored [67]. Clifford et al. [38] classified them as two main categories: *temporally ungrouped* and *temporally grouped*. The second representation is said to have more expressive power and to be more natural since it is history-oriented [38]. TSQL2 [96] tries to reconcile the two approaches [38] within the severe limitations of the relational tables. It needs to introduce new temporal constructs for a complete set of queries, which hinders commercial vendors from fully integrating it into their current DBMS products.

Object-oriented temporal models are compared in [77], and a formal temporal object-oriented data model is proposed in [21]. The problem of version manage-

ment in object-oriented and CAD databases has received a significant amount of attention. However, support for temporal queries is not discussed, although query issues relating to time multigranularity were discussed in [29].

Oracle implements Flashback [4], an advanced recovery technology that allows users to rollback to old versions of tables in case of errors. However, Flashback only provides limited queries, and efficiently support of temporal queries is not provided.

Recently Microsoft Research has developed a transaction database called Immortal DB. It implicitly embeds timestamps into transaction records, uses various lazy-timestamping and time-based indexing techniques, and supports snapshot queries of past database content. However, the system is mainly focused on transaction management, instead of query support. As a result, complex temporal queries, such as those described in the literature and supported in the proposed TSQL2 standard are not supported in Immortal DB.

2.2.3 Coalescing Query Support in Transaction Databases

To determine whether a certain approach is effective to support a temporal database application or not, one very important factor is its efficiency to support classical temporal queries, such as coalescing. No matter what data representation and query translation we choose, we still need a good implementation of basic coalescing queries.

Coalescing is a data restructuring operation applicable to temporal databases. It is similar to duplicate elimination in conventional databases. Coalescing merges timestamps of adjacent or overlapping tuples that have identical attribute values [22]. For instance, consider the snapshot of a temporal relation in Table 2.1 that shows data about employees working in a company, a new timestamped

EMPNO	SALARY	TITLE	DEPTNO	TSTART	TEND
1001	60000	Engineer	d01	1995-01-01	1995-05-31
1001	70000	Engineer	d01	1995-06-01	1995-09-30
1001	70000	Sr Engineer	d02	1995-10-01	1996-01-31
1001	70000	Tech Leader	d02	1996-02-01	1996-12-31

Table 2.1: The table EMPLOYEE_HISTORY

tuple is generated whenever there is a change in any of the attribute values. The well-know problem with this approach is that coalescing is needed when some of the attributes are projected out [96]. Much research has focused on this problem, and the solutions proposed include the TSQL2 [78] approach, and the point-based temporal model [84].

Suppose that the manager of the organization is interested in the history of the salary of employee 1001. The semantic of this query could be expressed using TSQL (Temporal SQL), which is an extension of the standard relational query language (SQL) enhanced with temporal features and predicates to manipulate temporal databases [78]. In such a query, coalescing is applied on both *EMPNO* and *SALARY* attributes, where *EMPNO* is the primary key of the temporal relation and *SALARY* is the time-varying attribute of the user interest. The TSQL statement of the above query looks as follows:

Example 2.4 *TSQL implementation to query the history of the salary of employee 1001.*

SELECT *EMPNO, SALARY*
FROM *EMPLOYEE_HISTORY (EMPNO, SALARY)*
WHERE *MPNO = 1001*

Applying coalescing to these four tuples generates the result shown in Ta-

EMPNO	SALARY	TSTART	TEND
1001	60000	1995-01-01	1995-05-31
1001	70000	1995-06-01	1996-12-31

Table 2.2: The result of Example 2.4

ble 2.2. Because the timestamps of the three tuples with a salary of 70000 are adjacent, coalescing merged the three tuples into a single tuple with new timestamps in Table 2.2. The *TSTART* value of the new timestamp is the *TSTART* value of the first tuple, where the *TEND* value is the *TEND* value of the last tuple.

In [22], Bohlen et al. study the problem and propose that coalescing could be implemented through either (i) SQL-based implementation, (ii) main memory implementation, or (iii) DBMS extensions. The DBMS extension approach requires modifying the underlying DBMS internals, which is something difficult and expensive. The main memory implementation approach works by loading a relation into main memory, coalesce it, and then store it back to the database. This approach suffers from two main problems. First, it might be impossible, in many cases, to load the whole relation in main memory. Second, it is an expensive task to periodically move a relation from the database to the running application and then store it back to the database. The SQL implementation approach aims at expressing coalescing operation as a set of SQL commands that runs on the database and generate a coalesced relation. However, usually the coalescing query is first very complex, and more seriously, it requires several scans, as well as, self-join(s) to the entire temporal relation. Therefore, efficient support for coalescing represents an important difficult problem for DBMSs. In Chapter 3, we will shortly review the idea of several alternatives to implement coalescing using SQL, and mainly discuss our approach using latest SQL:2003 standards and UDA to solve this problem.

2.3 Streaming XML Documents and Data Stream Management Systems

XML has been widely accepted as the standard data representation for information exchange on the web. In particular, XML stream systems have attracted much interest recently, because of its use in a wide range of applications including sensor networking, online auctions, purchase order processing, retail transaction management, publish/subcribe system, etc. In this section, we will summarize various research achievements in streaming XML processing and data stream management systems.

2.3.1 Stream-based XML Processing

In an emerging paradigm for XML query processing, XML data continuously arrives from external sources, and queries are evaluated every time when a new data item is received. Such XML query processing is referred to as stream-based. A distinctive feature of stream-based processing is the ability to process data as it arrives. One application of such streaming XML processing is for XML message brokering, where messages need to be filtered and transformed on-the-fly. Furthermore, in cases where incoming messages are large, stream-based processing also allows query execution to start long before those messages are completely received, thus reducing the delay in producing results.

2.3.1.1 Event-based XML Query Processing

Stream-based XML query processing can be performed at the granularity of a document or a smaller constituent piece of a document. Some of the earlier Continuous Query systems, such as NiagaraCQ [66], execute queries when new

documents arrive. Documents in these systems are simple and small, e.g., stock quote updates and event notifications. As XML gains popularity in a wide range of applications, XML documents have been used for encoding data of diverse types and immense sizes (e.g., the equivalent of a databases worth of data). To provide efficient processing also for such large documents, more recent systems such as XFilter [16], Tukwila [54], and the BEA Stream Processor [47] support fine-grained query processing upon arrival of a start-tag, end-tag, or text data of an element.

Fine-grained XML query processing can be implemented via event-based APIs. A well known example is the SAX interface [23], which reports low-level parsing events incrementally to the calling application. Figure 2.2 shows an example of how a SAX interface breaks down the structure of the sample XML document from Figure 2.1 into a linear sequence of events.

<Start Document
<Start Element: *book*
<Attribute: *id = "DC1001"*
<Start Element: *title*
Characters: *A Complete Guide to DB2 Universal Database*
>End Element: *title*
<Start Element: *author*
Characters: *Don Chamberlin*
>End Element: *title*
...
>End Document

Figure 2.2: SAX API for sample XML document book.xml

To use the SAX interface, the application receiving the events must implement handlers to respond to different events. In particular, stream-based XML query

processors can use these handlers to implement event-driven query processing.

2.3.1.2 Finite State Machine-Based Approach

A popular approach to event-driven XQuery processing has been to adopt some form of Finite State Machine (FSM) to represent path expressions [16, 54]. This approach is based on the observation that a path expression written using the axes ("/", "//") and node tests (element name or "*") can be transformed into a regular expression. Thus, there exists an FSM that accepts the language described by such a path expression [53].

Both XFilter [16] and Tukwila [54] create an FSM for each path expression by mapping the location steps of the path expression to machine states.

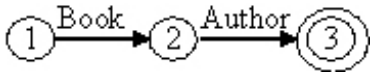

Figure 2.3: FSA for Q0: /Book/Author

Figure 2.3 shows an example FSM created for a simple path expression, where the two concentric circles represent the accepting state of this FSM. Arriving XML documents are then parsed with an event-based parser (e.g., a SAX parser). The events raised during parsing are used to drive the execution of query FSMs; in particular, "start element" events drive the FSMs through their various transitions, and "end element" events cause the FSMs to backtrack. A path expression is said to match a document if during parsing, the accepting state for that path is reached.

Many FSM-based techniques to answer queries on streaming XML input have been proposed, with different application foci. Techniques explored include single-query processing [54, 44], processing of multiple XPath expressions [95], process-

ing of XPath for twig-queries [32], predicate evaluation [51], backward-axes handling [19], and streaming queries with disk-resident index assistant [24].

2.3.2 Data Stream Management Systems

There has been a significant amount of activity on the topic of handling continuous, rapid, and time-varying tuple streams, which results in the development of a number of data stream management systems [31, 49, 63, 65]. The Tapestry project [18, 83] was the first to focus on the problem of 'queries that run continuously over a growing database'. Recent work in the Telegraph project [33, 65] focuses on efficient support for continuous queries and the computation of standard SQL-2 aggregates that combine streams flowing from a network of nodes. The Tribeca system focuses on network traffic analysis [81] using operators adapted from relational algebra. The OpenCQ [63] and Niagara Systems [37] support continuous queries to monitor web sites and similar resources over the network, while the Chronicle data model uses append-only ordered sets of tuples (chronicles) that are basically data streams [55]. The Aurora project [31] aims at building data management systems that provide integrated support for both data stream applications and stored data applications. The problem of designing a general-purpose data stream management system (DSMS) is the focus of the CQL [17].

The objective of overcoming SQL's limitation in dealing with time series and sequences has motivated significant database research long before the emergence of data streams, wherein SQL-TS [72, 73] is an extension of SQL where sequences and time-series, are searched using regular-language expression embedded in the SQL FROM clause. Similarly, the work in ATLaS [92, 61] is also aimed to provide a native extension to SQL query language, where user-defined aggregates (UDA) can be supported for complex data mining applications and streaming queries.

As we will see in this paper, UDA, which is a simple extension to SQL, can be easily used to support windows, time-series queries, and sequence queries on data streams.

2.4 OLAP Queries in XML

OLAP queries have been studied widely in business applications based on relational data model. [68] presents a survey of logical models for OLAP. Most database vendors support OLAP constructs such as "ROLLUP", "CUBE" in their products; [69] presents a detailed summary of industrial OLAP offerings.

Over the years, Online Analytical Processing (OLAP) has been studied extensively. We report here a few related efforts. Vassiliadis and Sellis [68] present a survey of logical models for OLAP databases. Gray et al. [48] first introduced the OLAP CUBE operator. Chaudhuri and Dayal [35] present an overview of relevant concepts in data warehousing and decision support systems. Most database vendors support OLAP in their database systems. The OLAP Report [69] presents a detailed study of current industrial OLAP offerings. Current work in using XML for OLAP applications involves using XML for representing external data. To the best of our knowledge, no one has investigated exploiting XML's tree model for analytical purposes. Recently, Pedersen et al. have been exploring the integration of XML data with the traditional OLAP processing [40]. Jensen et al. describe how to specify multi-dimensional OLAP cubes over source XML data [56].

However, XQuery, the major query language of XML data, does not include an explicit grouping construct comparable to the group by clause in SQL. Lack of an explicit grouping construct makes certain classes of common business ana-

lytic queries needlessly difficult to express and execute efficiently. Furthermore, business objects often have complex structures and varying schemas demanding advanced analytic capabilities that are even more difficult to express in the current XQuery syntax. In this section, we will review the grouping problem in XML, and summarize the recent approaches to enable XQuery with OLAP queries.

2.4.1 Grouping Problems in XML

We motivate the problem with an illustrative example. The example query is based on an input document with similar schema as Figure 2.1. To make it more general, a book may have zero or more authors and zero or one publisher.

Suppose that we need to find the average price of books for each publisher and year. This query may be expressed in the current XQuery syntax, as follows:

Example 2.5 *(Current XQuery Implementation) Find the average price of books for each publisher and year*

```
FOR $a IN distinct-values(//book/publisher)
        $b IN distinct-values(//book/year)
LET $p := //book[publisher = $a and year = $b]
WHERE fn:exists($p)
ORDER BY $a, $b
RETURN
    <group>
        <publisher> $a </publisher>
        <year> $b </year>
        <avg-price> avg($p) </avg-price>
    </group>
```

This XQuery expression involves computing the set of distinct publishers and the set of distinct years, then finding the set of books corresponding to each (publisher, year) pair and computing the average net price for each non-empty set of books. Arguably, this is not the most natural way for users to express the intent of grouping books by publisher and year. More seriously, a straightforward execution of this query would be very inefficient because it would involve many passes over the set of books resulting in expensive and redundant navigation and self-joins. In order to produce an efficient plan, an XQuery optimizer would need to detect the grouping implied by the query, which might be quite difficult to do in complex cases. The expression shown above also suffers from the problem of missing rows for books that do not have any publishers. This is a consequence of the fact that the non-existence of a *publisher* element for a book will not be represented in the sequence of *publisher* elements produced by the expression *//book/publisher* in the first *For* clause.

2.4.2 OLAP Support on XML Data

Complex OLAP extensions on hierarchies defined over relational data have been studied in [25]. To support OLAP queries for specific XML data model, [40] has some exploration on the integration of XML and traditional OLAP processing, and [64] presents a study on how to specify cube queries on XML data. However, none of the above focus on query language for specifying the OLAP queries.

[14, 58] did quite a lot of research in the area of supporting grouping functionalities in XQuery, by analyzing some native problems of XML data model for support grouping queries. Natix [82] provides a tuple-based algebra that includes grouping operators for construction of XML elements.

One major research proposal is provided in [70, 57], where GROUP BY constructs to better support OLAP applications and analytics are introduced into XQuery standard.

According to [70]'s proposal, a *Group By* clause and a *Nest* clause are introduced to current XQuery's FLWOR structure. The expressions in the *Group By* clause (immediately following the *Group By* keyword) are called grouping expressions and the variables they are bound to are called grouping variables. The expressions in the *Nest* clause are called nesting expressions and the variables they are bound to are called nesting variables. Each tuple in the output stream represents one group and consists of a value for each grouping variable and each nesting variable.

As an example, the query in Example 5.1 can be expressed with the following extended XQuery.

Example 2.6 *(XQuery with Group By Implementation) Find the average price of books for each publisher and year.*

FOR *$a* IN *//book*
GROUP BY *$a/publisher* INTO *$p, $a/year* INTO *$y*
NEST *$a/price* into *$prices*
RETURN
<group>
{$p, $y}
<avg-price> avg($prices) </avg-price>
</group>

In the above query, in the input tuple stream that is seen by the *Group By* clause, each tuple contains one variable: $b, bound to one book. In the output

tuple stream generated by the group by clause and used by the return clause, each tuple contains three variables: $p, bound to a publisher element, $y, bound to a year element, and $prices, bound to a sequence of atomic values that represent prices of books.

For this simple grouping problem, this *Group By* approach provides a decent solution. But we will further study in Chapter 5 to see what problems lie in this method, and how we solve the OLAP query in XML query language by function import mechanism.

CHAPTER 3

Managing Historical Information using XML and Relational Databases

While the introduction of temporal extensions into database standards has proven difficult to achieve, the newly introduced SQL:2003 and XML/XQuery standards have actually enhanced our ability to support temporal applications in commercial database systems. We illustrate this point by discussing three approaches that use temporally grouped representations. We first compare the approaches at the logical level using a common set of queries on an XML-based historical view; then we turn to the physical level and discuss our ArchIS system that supports the three different approaches efficiently in one unified physical implementation. We also provide an algorithm which can implement temporal coalescing queries very efficiently, and use SQL:2003 and UDAs to express it. We conclude that the approaches of managing transaction-time information using XML and SQL can be integrated and supported efficiently within the current standards.

We seek to support historical information management and temporal queries without extending current standards. Our insistence on using only current standards is inspired by the lessons learned from the very history of temporal databases, where past proposals failed to gain much acceptance in the commercial arena, in spite of great depth, breadth [67, 50] and technical elegance [78, 80]. Here, we simply accept the fact that temporal extensions to existing standards

are very difficult to sell, in spite of the growing pull by temporal applications; then, we move on from there by exploring solutions that do not require extending current standards. This low-road approach is hardly as glamorous as the "new temporal standards" approach pursued in the past, but it is not without interesting research challenges and opportunities, as we will show in this chapter. In particular, new opportunities are offered by two recent developments that have taken information systems well beyond SQL:1992 which, in the past, supplied the frame of reference for temporal database research. The first development is the introduction of XML/XQuery standards, that have gained wide acceptance by all DBMS vendors, and the second is the introduction of SQL:2003 [1, 15, 7], which contains new advanced features such as nested relations and OLAP functions.

The benefits of XML in temporal information management include: i) XML can be used to represent data in a temporally grouped data model, ii) XQuery provides an extensible and Turing-complete language [60], where new temporal functions can be defined in the language itself.

These features make it possible to use XML to represent the history of relational databases by timestamping the grouped attribute histories of each table, and XQuery to express complex temporal queries [88, 90, 87, 45]. This approach requires no extension to current standards, and it is very general, insofar as it can be used to represent and query the transaction-time, valid-time and bitemporal history of databases [87], and arbitrary XML documents [86]. Therefore, contrasting this experience with the past one focusing on SQL, we might simply conclude that XML and its query languages are more supportive to historical information management and temporal queries than SQL.

However, there are many reasons for which we are not prepared to give up on SQL. Indeed, relations represent a simple and intuitive data model which comes

with (i) a built-in graphical representation in form of tables, (ii) WYSIWYG query languages such as QBE, and (iii) unique areas of commercial strength, such as OLAP applications and data warehousing. By contrast, (i) there is no built-in graphical rendering for an XML document, and this must be provided by the user via stylesheets, (ii) WYSIWYG XML query languages require further research and development, and (iii) XQuery is more complex than SQL, and its commercial application areas are still emerging.

There is also the critical issue of performance. In particular, in supporting transaction-time history of relational databases in XML [90, 45], we compared the two approaches of (i) implementing temporal queries directly in a native XML system, and (ii) recasting these views into historical tables, whereby the original XQuery statements are then mapped into equivalent SQL (or SQL/XML [8]) queries. Our experiments show that the second approach tends to be significantly more efficient [90, 45].

Therefore, both logical and physical considerations point to the conclusion that SQL is to remain the database language of choice, for a long time to come— particularly in data warehousing and business-intelligence applications— and every effort should be made to assure efficient management for historical information and temporal queries in SQL:2003. Toward this goal, we will take full advantage of the lessons learned in supporting temporal queries in XML and seek an efficient support for temporal queries independent of whether they are expressed in SQL or XQuery.

empno	name	salary	title	deptno	start	end
1001	Bob	60000	Engineer	d01	1995-01-01	1995-05-31
1001	Bob	70000	Engineer	d01	1995-06-01	1995-09-30
1001	Bob	70000	Sr Engineer	d02	1995-10-01	1996-01-31
1001	Bob	70000	TechLeader	d02	1996-02-01	1996-12-31

Table 3.1: The snapshot history of employees

3.1 Historical Database Modeling with XML

3.1.1 Temporally Grouped Data Model Using XML Representation

To achieve a concrete realization of the temporally grouped data model, we propose an XML based approach to represent the temporal information.

Suppose we have the history of employees and departments as they would be viewed in traditional transaction-time databases [67] using a temporally ungrouped representation. Table 3.1 shows the corresponding table schema, where `empno` is the key.Here we will use temporally-grouped representations that exploit the expressive power of XML and its query languages. Thus, instead of the representation shown in Table 3.1 we will use the representation shown in Figure 3.1. We will call it H-view when it is virtual representation. Each element in an H-View is assigned the attributes `tstart` and `tend`, to represent the inclusive time-interval of the element. The value of `tend` can be set to *now*, to denote the ever-increasing current time. Note that there is an intrinsic constraint that the interval of a parent node always covers that of its child nodes. The H-view also has a simple and well-defined schema.

```
<employees tstart="1995-01-01" tend="1996-12-31">
 <employee tstart="1995-01-01" tend="1996-12-31">
  <empno tstart="1995-01-01" tend="1996-12-31">1001</empno>
  <salary tstart="1995-01-01" tend="1995-05-31">60000</salary>
  <salary tstart="1995-06-01" tend="1996-12-31">70000</salary>
  <title tstart="1995-01-01" tend="1995-09-30">Engineer</title>
  <title tstart="1995-10-01" tend="1996-01-31">Sr Engineer</title>
  <title tstart="1996-02-01" tend="1996-12-31">Tech Leader</title>
  <deptno tstart="1995-01-01" tend="1995-09-30">d01</deptno>
  <deptno tstart="1995-10-01" tend="1996-12-31">d02</deptno>
 </employee>
</employees>
```

Figure 3.1: H-view: XML-based Representation of Employees' History

3.1.2 Temporal Queries in XML

A full spectrum of queries was presented in [90] to illustrate the effectiveness of the approach—including temporal projection, temporal snapshot, temporal slicing, temporal join, and temporal aggregate. Because of space limitations, we restrict ourselves to the following three examples that will be used throughout this chapter.

Example 3.1 *Temporal Projection. Retrieve the title history of employee "1001".*

Element title_history{
FOR *$t* IN *doc("employees.xml")/employees/employee[empno="1001"]/title*
RETURN *$t* }

Observe that no coalescing is needed after this projection, since the history of titles is temporally grouped.

Example 3.2 *Temporal Snapshot. Retrieve titles and salaries of all employees on 1994-05-06.*

FOR *$e* IN *doc("employees.xml")/employees/employee*
LET *$t := $e/title[tstart(.) <= date("1994-05-06") and*
tend(.) >= date("1994-05-06")]
LET *$s:=$e/salary[tstart(.) <= date("1994-05-06") and*
tend(.) >= date("1994-05-06")]
RETURN *<employee>{$e/empno,$t,$s}</employee>*

In this query, we need to check only the timestamps of the leaf nodes, since the H-document has a *temporal covering constraint*, i.e., the interval of a parent node always covers that of its child nodes.

Here, *date()* is a built-in function of XQuery (for simplicity, we omit the namespace *fn*). Instead, *tstart()* and *tend()* are user-defined functions to shield the user from the complexity of the underlying representation, since, e.g., 'now' [39] requires special representation and special handling (in ArchIS [90] we use the end-of-time to represent 'now'). *date()* is a built-in function of XQuery (for simplicity, the namespace *fn* is omitted here).

Example 3.3 *Retrieve the salary history of employees in dept. "d01", while they were in that department.*

FOR *$e* IN *doc("employees.xml")/employees/employee[deptno="d01"]*
FOR *$s* IN *$e/salary*
FOR *$d* IN *$e/deptno[.="d01"]*
LET *$ol := overlapperiod($s, $d)*
WHERE *not (empty($ol))*
RETURN

element sal_history {

element empno $e/empno/text(),

element salary$s/text(),

$ol

}

Here *overlapperiod($a, $b)* is a user-defined function that returns an element *PERIOD* with overlapped period as attributes *(tstart, tend)*; if there is no overlap, then no element is returned which satisfies the XQuery built-in function *empty()*.

3.1.3 Discussion

The previous examples illustrate that XQuery is capable of expressing complex temporal queries, but the expression of these queries can be greatly simplified by a suitable library of temporal functions. For instance, the ArchIS system [90] we have implemented in UCLA supports a rich set of functions, including the simple scalar functions described above, and also complex functions, including temporal aggregates and coalesce functions.

A significant benefit offered by the XML/XQuery-based approach to temporal information management is that it is very general and can handle the history of arbitrary XML documents that have evolved through successive versions [86]. The approach can also be extended to valid-time databases and bitemporal databases [87].

On the other hand, the ease of use of XQuery is questionable, and the problem of displaying the results of temporal queries in user-friendly ways can be a real challenge, since the tagged representations, such as that of Figure 3.1, are not suitable for casual users. To produce visually appealing representations, the query

designer might have to code a stylesheet, using XSL [9]—possibly a different one for each query. This problem is far from trivial, and the visual rendering of temporal information poses interesting research challenges.

3.2 Nested and Rollup Relational Table Implementation

The growing popularity of XML in web-oriented applications does not change the fact that SQL remains the cornerstone of database applications, and its importance in areas such as business intelligence and data warehouses is growing every day. For these reasons, efficient support for temporal information and queries in SQL remains critical. Therefore, we explore two approaches: one based on nested relations, which is discussed in Section 3.2.1, and another based on OLAP tables, which is discussed in Section 3.2.2.

3.2.1 DB History and Nested Relations

Nested relations are part of the latest SQL:2003 standards, and also supported by most commercial database vendors [7, 3]. Therefore a temporally grouped representation, similar to that used with XML, can also be achieved within SQL standard. For instance, for our *employee_history* example, we can use the following DDL to create the schema containing the nested table ('n-table' for short, or 'n-view' if it is a nested view) *n_employee*:

Example 3.4 *DDL to create nested table n_employee.*

```
CREATE TYPE salary_typ AS OBJECT(
    salary NUMBER(7),
    timep PERIOD
    );
```

```
CREATE TYPE salary_tbl AS TABLE OF salary_typ;
...
CREATE TABLE n_employee(
    empno VARCHAR2(8),
    timep PERIOD,
    n_name name_tbl,
    n_salary salary_tbl,
    n_title title_tbl,
    n_deptno deptno_tbl)
NESTED TABLE n_salary STORE AS n_salary,
NESTED TABLE n_title STORE AS n_title,
NESTED TABLE n_deptno STORE AS n_deptno;
```

This definition uses the user-defined type *PERIOD*, which can be defined in SQL:2003 as follows:

```
CREATE TYPE PERIOD AS OBJECT(
    tstart DATE,
    tend DATE
    )
```

The same temporal queries that we have expressed on XML using XQuery can now be expressed on nested tables using SQL:2003, as follows:

Example 3.5 *History projection. Retrieve the title history of employee "1001" (Example 3.1 on nested relations)*

```
SELECT t.*
FROM n_employee e, TABLE(e.n_title) AS t
WHERE e.empno='1001'
```

Example 3.6 *Temporal Snapshot. Retrieve titles and salaries of all employees on 1994-05-06 (Example 3.2 on nested relations)*

SELECT *t.title, s.salary*
FROM *n_employee e, TABLE(e.n_title)* AS *t, TABLE(e.n_salary)* AS *s*
WHERE *tstart(t.timep) <= '1994-05-06'*
AND *tend(t.timep) >= '1994-05-06'*
AND *tstart(s.timep) <= '1994-05-06'*
AND *tend(s.timep) >= '1994-05-06'*

Here too we use the functions *tstart()* and *tend()* to isolate the user for the internal representation of time, including 'now'. (Support for user-defined scalar functions is now available in all commercial OR-DBMSs.)

Example 3.7 *Retrieve the salary history of employees in dept."d01", while they were in that department (Example 3.3 on nested relations)*

SELECT *e.empno, overlapperiod(d.timep, s.timep), s.salary*
FROM *n_employee* AS *e, TABLE(e.n_dept)* AS *d,*
TABLE (e.n_salary) AS *s*
WHERE *d.deptno = 'd01'* AND *overlaps(d.timep, s.timep)*

Here *overlaps()* is defined to return true if two periods overlap, and false otherwise; *overlapperiod()* is defined to return the overlapped *PERIOD*.

In addition to scalar functions, such as *overlapperiod()*, temporal aggregates (e.g., the temporal version of min and sum [78]) will be required by temporal queries. These new functions could be easily built into commercial systems by the vendors, or by the users, since commercial OR-DBMSs now support the introduction of new scalar and aggregate functions coded in a procedural language.

(In the ATLaS system [92], user-defined aggregates can also be introduced natively in SQL, with no recourse to external PLs.) The new temporal aggregates that must be introduced include, the *rising* function of TSQL2 [78], and also the *tcoalesce* aggregate for temporal coalescing— since the temporally grouped representation made possible by nested tables has greatly reduced the need for coalescing, but not eliminated it all together (and the same is true for XML).

Assuming that a library containing the basic temporal functions is available, the complexity of writing temporal queries in SQL:2003 and nested tables is about the same as writing them in XQuery and XML. Both approaches present users with more alternatives in presenting data than flat relations. For instance, the join of nested employees and departments tables can be represented by a one-level hierarchy where the department and employee attributes are at the same level, or as a hierarchy where employees are grouped inside departments, or vice-versa. We next return to the 'Spartan simplicity' of flat relations, in which the alternatives are fewer and the problem is simplified.

3.2.2 An OLAP-Inspired Representation

A temporally grouped representations can also be obtained by using null values in flat tables such as those returned by OLAP aggregates. Thus, the transaction-time history of employees, that was described by tuple timestamping in Table 3.1, and as an XML document in Figure 3.1, is now described as a flat table with null values as shown in Figure 3.2, where the null value is represented by the question mark, "?". This representation can be defined by a ROLLUP operation on Table 3.1, defined by the following SQL statement (again here we use *timep* to represent the period of *tstart* and *tend*)[1].

[1]As in the case of OLAPs, we might also want to represent the null values generated by the rullup operation differently from those representing null values in the original table.

empno	tstart	tend	salary	title	deptno
1001	1995-01-01	1996-12-31	?	?	?
1001	1995-01-01	1995-05-31	60000	?	?
1001	1995-06-01	1996-12-31	70000	?	?
1001	1995-01-01	1995-09-30	?	Engineer	?
1001	1995-10-01	1996-01-31	?	Sr Engineer	?
1001	1996-02-01	1996-12-31	?	Tech Leader	?
1001	1995-01-01	1995-09-30	?	?	d01
1001	1995-10-01	1996-12-31	?	?	d02

Figure 3.2: The history view e_employees

Example 3.8 *DDL to create event-history tables e_employee.*

CREATE VIEW *e_employee* AS

SELECT *empno, tcoalesce(timep,salary,title,deptno)*

FROM *employee_history*

GROUP BY GROUPING SETS

(empno,(empno,salary),(empno,title),(empno,deptno))

We refer to the representation shown in Figure 3.2 as an 'e-table' (or 'e-view' if it is a view) because this captures the event-history for employees, as it will be discussed in Section 3.2.2.1. Moreover, temporal queries on e-tables preserve the traditional style of SQL queries:

Example 3.9 *History projection. Retrieve the title history of employee "1001" (Example 3.1 on event-history relations)*

SELECT *title, tstart, tend*

FROM *e_employee*

WHERE *e.empno='1001'* AND *title IS NOT NULL*

Example 3.10 *Temporal Snapshot. Retrieve titles and salaries of all employees on 1994-05-06 (Example 3.2 on nested relations)*

SELECT *e.empno, e.title, e.salary*
FROM *e_employee* AS *e*
WHERE *tstart(e.timep) <= '1994-05-06'*
AND *tend(e.timep) >= '1994-05-06'*
AND *e.title IS NOT NULL* OR *e.salary IS NOT NULL*

This query assumes that we only want to retrieve the information, without reformatting it. However, if we want to reformat the information derived into a join table, then we also want to join the titles and salaries of all employees at that date into a flat relation as follows:

SELECT s.empno, t.title, s.salary
FROM e_employee AS s, e_employee AS t
WHERE tstart(s.timep) <= '1994-05-06' AND tend(s.timep) >= '1994-05-06'
AND tstart(t.timep) <= '1994-05-06' AND tend(t.timep) >= '1994-05-06'
AND s.empno = t.empno
AND t.title IS NOT NULL AND s.salary IS NOT NULL

Example 3.11 *Retrieve the salary history of employees in dept. "d01", while they were in that department (Example 3.3 on event-history relations)*

SELECT *n1.empno, n1.salary, overlapperiod(n1.timep,n2.timep)*
FROM *e_employee* AS *n1, e_employee* AS *n2,*
WHERE *n1.empno = n2.empno*
AND *n1.salary IS NOT NULL* AND *n2.salary IS NOT NULL*
AND *d.deptno = 'd01'* AND *overlaps(n1.timep, n2.timep)*

This query illustrates the use of temporal joins, with intersection of overlapping periods; these are required for query Q3 in all three representations. While the complexity of queries is similar for our three temporally-grouped approaches, e-tables offer unique advantages that are discussed next.

3.2.2.1 Event-Oriented Histories

An advantage of this last representation is that grouping can be easily controlled by the ORDER BY clause in SQL. For instance, the representation of Figure 3.2, where the history of each employee attribute is grouped together, is produced by the following clause:

SELECT *empno, timep, salary, title, deptno*
FROM *e_employee*
ORDER BY *empno,salary,title,deptno, tstart(timep)*

Since the null value is assumed to be the last value in each domain, this ORDER BY clause indeed produces the table of Figure 3.2.

Assume now that we want to view the history of events, pertaining to employees' salaries and departments, that have occurred in the company; then we can just list them in ascending chronological order as follows:

SELECT *timep, salary, title, deptno*
FROM *e_employee*
ORDER BY *tstart(timep)*

Assume now that we want to view the personnel history of the various departments. Then, we would simply use the following ORDER BY clause:

SELECT *deptno, timep, empno, salary, title*
FROM *e_employee*
ORDER BY *deptno,tstart(timep),empno,salary,title*

This last statement returns all the events grouped by department and arranged in chronological order; moreover if we want to see the history of the personnel in each department grouped by individual employees, we only need to switch the order of *empno* and *tstart(timep)*.

The visual presentation of historical data and query results is much simpler using e-tables than using n-tables or H-tables (which is discussed later in Section 3.3.1). This is because flat tables come with their built-in graphical representation, while, e.g., XML requires the user to write a style sheet to visualize data. Moreover, as demonstrated by the previous examples, restructuring on e-tables can be realized by simply reordering the tuple using an ORDER BY clause, whereas it might require complex nesting and unnesting in the other representations.

In most temporal database approaches, including TSQL2 [78], a temporal relation can be either declared as a state table or as an event table but the two views are not easily combined. A simple mapping between the two views is highly desirable since, in everyday life, states and events are two facets of the same evolving reality. Moreover, many advanced applications, such as time-series analysis [34], sequence queries [71], and data stream queries [61], view the database as a sequence of events, rather than a sequence of states.

The e-tables just described, make it possible the unification of state-based

and event-based representations by simply using SQL ORDER BY construct. For instance, say that we want to find employees who have been transferred from a department to another, and from this, back to the old one. To answer this query by perusing the history of employees, we would probably start by carefully viewing the results of the following query:

Example 3.12 *Reordering to detect round-trip transitions between departments.*

SELECT *empno, timep, depno, salary, title*
FROM *e_employee*
ORDER BY *empno, tstart(timep)*

Then, the immediate sequence of any three tuples with non-null *deptno* column, would satisfy the query—provided that the first department is equal to the third (and that there was no interruption in the employee's employment).

Although this query is conceptually simple, it requires the detection of three successive tuples—an operation that is rather complex and inefficient to express in standard SQL. A first solution to this problem is to write a user-defined aggregate (UDA); in fact UDAs can easily express state-based computations [91]. Moreover, several event-patterns and sequence languages for time-series analysis have been proposed in the literature [75, 34, 71] and would work very nicely with the representation discussed here. For instance, using SQL:2003 our query could be expressed as follows:

Example 3.13 *From department A to B and back, with no other change in between.*

SELECT *empno,*
FIRST*(deptno)* OVER 2 ROWS PROCEDING ORDER BY *tstart(timep) A*
FIRST*(deptno)* OVER 1 ROWS PROCEDING ORDER BY *tstart(timep) B*

LAST*(deptno)* OVER 1 ROWS PROCEDING ORDER BY *tstart(timep) C*
WHERE $A = C$ AND B *IS NOT NULL*
GROUP BY *empno*

Here, the FROM clause specifies that, given the ordering described above, A, B and C are three successive tuples that are also related by the conditions specified in the WHERE clause. Space limitation prevents us from delving into languages as SQL-TS [71], although they represent a very interesting and pertinent topic in temporal database research. Here, it suffices to observe that these languages rely on tuples being arranged in a suitable order—which is easier to achieve with e-tables than with H-tables or n-tables.

3.3 Efficient Implementation

In the previous sections, we have discussed the pros and cons of alternative representations for temporal history. In reality, these are likely to be supported together, rather than as alternatives, since database vendors are gung ho on supporting both SQL and XML in their systems. Practical considerations also suggest that a unified implementation at the internal level should be provided for these multiple external views. At UCLA, we have been developing the ArchIS system that unifies the support for multiple external temporal models into one architecture [90, 45].

The basic architecture of ArchIS [90] is shown in Figure 3.3. ArchIS is designed to preserve and archive the history of the database by preserving the evolution of its content, either by using active rules attached to the database or by periodically visiting their update logs. ArchIS then supports alternative logical views of the database history described in the previous sections, by mapping queries against

these views into equivalent queries against the history database. In our previous work on the implementation of storing H-documents [90, 45], we have compared the use of a native XML DBMS such as the Tamino XML Server [74], against the approach of shredding these documents and storing them into RDBMSs. The second approach was found to offer substantial performance advantages and will be used here. (In our implementation, the 'current database' and the archived one are managed by the same system. But the results are easily generalized to the situations where these two are separate and even remote.)

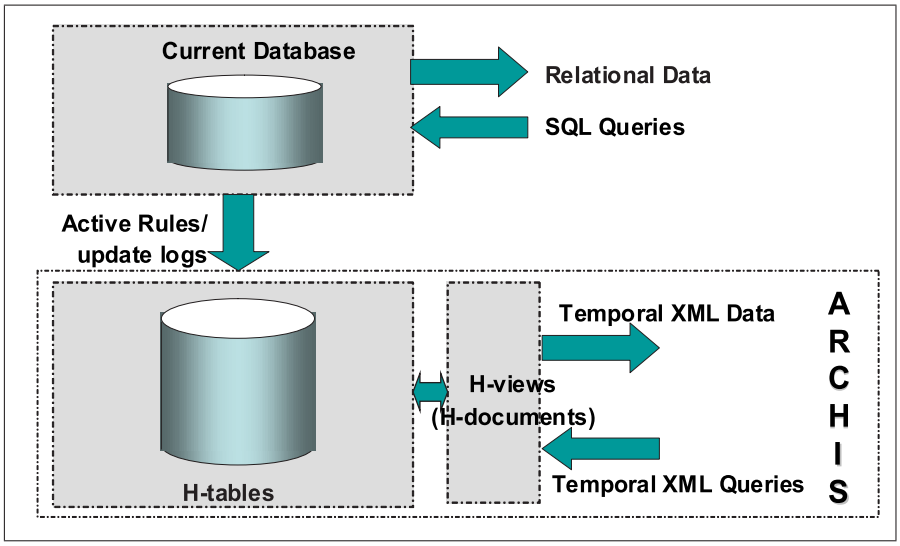

Figure 3.3: ArchIS: Archival Information System

In the next sections we first discuss the structure of the Key & Attribute History Tables used at the internal level, and then we describe the problem of mapping external queries into internal ones. We finally describe the temporal clustering and indexing techniques used in improving the performance of such queries.

The problem of supporting XML views through stored RDBMS tables is hardly new since it has recently provided a major focus for database research

[30, 41, 46]. However, here we do not need to support all XML documents and queries, but only historical views of database tables and temporal queries on such tables; thus, specialized techniques can be used for more efficient storage, and optimized query mapping.

3.3.1 History Tables

The history of each relation is preserved by a set of tables: one table for each attribute, and an additional table for the primary key of the original relation. Each tuple in the tables is timestamped with the two attributes *tstart* and *tend*. For example, consider our evolving DB relation

employee(empno, salary, title, deptno)

with key *empno*. The history of *employee* is preserved by following tables in ArchIS:

The Key Table:

employee_empno(empno, tstart, tend)

Since *empno* will not change along the history, the period *(tstart,tend)* in the key table also represents the valid period of the employee. The use of keys is for easy joining of all attribute histories of an object such as an employee.

Attribute History Tables:

employee_salary(empno, salary, tstart,tend)

employee_title (empno, title, tstart,tend)

employee_deptno(empno, deptno, tstart,tend)

The values of *empno* in the above tables are the corresponding key values, thus indexes on such *empno* can efficiently join these relations.

When a new tuple is inserted, the *tstart* for the new tuple is set to the current

timestamp, and *tend* is set to *now*. When there is a delete on a current tuple, we simply change the *tend* value in that tuple as current timestamp. An update can be viewed as a delete followed by an insert. We will later refer to these as key & attribute history tables (H-tables for short). H-tables could also be viewed as yet another candidate representation at the logical level; we have not considered them here because they do not provide real query advantages with respect to e-tables, and they make tasks such as reordering and visualization harder.

In addition to these, we also store information about the schema in a global relation:

relations(relationname, tstart, tend)

Our design builds on the assumption that keys (e.g., *empno*) remain invariant in the history. Otherwise, a system-generated surrogate key can be used.

3.3.2 Query Mapping

Mapping from e-views to H-tables. Mapping from H-tables to the e-views (or e-tables) of Figure 3.2 is simple, since the latter can be obtained taking the union of the H-tables after padding them with null values. This simple correspondence simplifies the translation and optimization of queries expressed on e-tables into equivalent queries on H-tables. The pattern of null values associated with the query plays an important role in the translation. Take for instance QUERY Q1e. The condition that *title IS NOT NULL* implicitly determines that salary and department must be null, and attribute table *employee_title* will appear in the WHERE condition of mapped query. Thus our original query is translated into:

SELECT *T.title, T.tstart, T.tend*
FROM *employee_title as T*
WHERE *T.empno = '1001'*

However, this is only the first step of the translation performed by ArchIS which also adds conditions to exploit the temporal clustering and indexing discussed later.

Mapping from n-views to H-tables. In DBMS that support nested relations, n-views (or n-tables) can be supported directly at the physical level. But even so, we might prefer to 'shred' and store them into flat H-tables, to simplify support for alternative external views (in particular, e-views), of for performance reasons, e.g., to take advantage of the clustering techniques available for H-tables, that will be discussed later. A simple approach to achieve this is to define a nested object-view (as defined in SQL:2003) on H-tables, as follows:

Example 3.14 *Define a nested object-view on H-tables.*

```
CREATE VIEW n_employee OF employee_t
WITH OBJECT IDENTIFIER (empno) AS
   SELECT e.empno, PERIOD(e.tstart, e.tend) AS timep,
      CAST(MULTISET(
         SELECT s.salary, s.tstart, s.tend
         FROM employee_salary s
         WHERE s.empno = e.empno)
         AS salary_tbl)
      ) AS n_salary,
   ...
   FROM employee_empno e;
```

With such a mapping, temporal queries on n-views are automatically trans-

lated by the DBMS into queries on H-tables through view definitions.

Mapping from XML-views to H-tables. The mapping from XML-views (or H-documents) to H-tables is significantly more complex. The problem of supporting XQuery on H-tables is similar in the sense that we have to generate efficient SQL queries, but more complex insofar as XML documents must be structured as output. Therefore, we use SQL/XML [8], whereby the results of SQL queries can be efficiently assembled into XML documents for output. Many database vendors now support efficient SQL/XML implementations, in which tag-binding and structure construction are done inside the relational engine for best performance [76]. In ArchIS [90], we compile XQuery statements on temporal XML-views, and optimize their translation into SQL/XML on the H-tables in five main steps, as follows:

1. *Identification of variable range.* For each distinct *tuple* variable in the original query, a distinct tuple variable is created in the FROM clause of the SQL/XML query, which refers to a certain key table or attribute table.
2. *Generation of join conditions.* There is a join condition `T.empno` and `N.empno` for any pair of distinct tuple variables.
3. *Generation of the WHERE conditions.* These are the conditions in WHERE clause of XQuery or specified in the XPath expressions.
4. *Translation of built-in functions.* The built-in functions (such as *overlaps($a,$b)*) are simply mapped into the corresponding SQL built-ins we have implemented for ArchIS.
5. *Output generation.* This is achieved through the use of the XMLElement and XMLAgg constructs defined in SQL/XML [8].

For instance, the SQL/XML translation of Example 3.1 is:

Example 3.15 *SQL/XML translation of Example 3.1.*

SELECT *XMLElement (Name "title_history",*
XMLAgg (XMLElement (Name "title",
XMLAttributes (T.tstart AS *"tstart", T.tend* AS *"tend"), T.title)))*
FROM *employee_title* AS *T*
WHERE *T.empno = '1001'*

3.3.3 Clustering and Indexing

Efficient support for historical queries requires support for temporal clustering and indexing; in ArchIS, this is achieved by a simple usefulness-based scheme whereby the H-tables are partitioned into segments [90]. For each table, the *usefulness* of its current segment is defined as the percentage of the segment tuples that have not expired yet (i.e., whose *tend* timestamp is still 'now'). The usefulness of the current segment is monotonically decreasing with time, and as soon as it falls below a user-specified percentage, the whole segment is archived, and a new segment is started containing only those tuples whose timestamps are 'now'. The segment number then can become part of the search keys supported by the indexes used in the database.

Thus, a request to find the salary of a given employee at certain time, could involve finding the corresponding segment in a small memory-resident index, and then using the (*segment_no, empno*) pair in the index search.

This usefulness-based scheme achieves temporal clustering through redundancy. Since there is no update in the archived tuples of a transaction-time database (unlike valid-time databases), redundancy does not generate additional execution costs. For reasonable usefulness values the extra storage costs are modest (e.g., 30% storage overhead for 33% usefulness [90]); this cost represents a

minor drawback, because of the fast decreasing cost of storage, and the applicability of compression techniques which has been proven in [90]. (The cost of re-compressing after updates is not present for archived data, since these are not updated.) On the other hand, the usefulness-based approach expedites archival search in a predictable and controllable fashion. For instance, for usefulness of 33% (1/3) we are assured that, when searching in the corresponding segments for records with a given timestamp, at least one of the three records visited has the right timestamp. Therefore, the time required to regenerate the past snapshot of a relation can be expected to be less than three times of that needed to generate the current snapshot from the current database [90]. Also, observe that the joining of H-tables require little extra time since they are already sorted on *empno*. The architecture and performance of ArchIS is covered in [90].

3.3.4 Summary

The realization of ArchIS system confirms the practicality of supporting both SQL-based and XML-based temporal views and queries with a unified and efficient internal representation. ArchIS can now run on top of IBM DB2 and the Stream Mill system [28]. We are currently working on extending it to run on commercial DBMS that support nested relations [7, 3], and explore any performance improvement that can be gained with this approach. We also plan to experiment with additional storage structures, such as R-trees, to better support valid-time and bitemporal databases.

3.4 Efficient Support for Coalescing Query

3.4.1 Problem Review: Support Coalescing with Pure SQL Queries

There are several alternatives to implement coalescing using SQL, either through SQL/PSM, cursors, or entirely in SQL [79]. The first two solutions require either an external programming module or in-memory cursors, which are very inefficient to support due to the high I/O manipulation cost. However, implementing coalescing entirely in SQL has the problem that the coalescing query is considerably complex and often requires multiple nested NOT EXISTS clauses [96], as well as, self-join(s). For instance, expressing Example 2.4 entirely in SQL can be achieved through the query shown in Example 3.16. Note that this query requires 6 scans, as well as, several self-joins on the entire temporal relation.

Example 3.16 *SQL implementation of coalescing Query as in Example 2.4.*

```
WITH Temp(Salary, TSTART, END) AS
    (SELECT SALARY, TSTART, END
    FROM EMPLOYEE_HISTORY
    WHERE EMPNO = 1001 )
  SELECT DISTINCT F.Salary, F.TSTART, F.TEND
  FROM Temp AS F, Temp AS L
  WHERE F.TSTART < L.TEND AND F.Salary = L.Salary
  AND NOT EXISTS
      (SELECT * FROM Temp AS M
      WHERE M.Salary = F.Salary
      AND F.TSTART < M. TSTART AND M.TSTART < L.TEND
      AND NOT EXISTS
          (SELECT * FROM Temp AS T1
```

```
        WHERE T1.Salary = F.Salary
        AND T1. TSTART < M. TSTART AND M.TSTART <= T1.TEND)
        )
AND NOT EXISTS
    (SELECT * FROM Temp AS T2
    WHERE T2.Salary = F.Salary AND
    ((T2. TSTART < F. TSTART AND F.TSTART <= T2.TEND)
    OR
    (T2.TSTART < L.TEND AND L.TEND < T2.TEND)))
```

Other alternatives to implement coalescing entirely in SQL is to use COUNT aggregate instead of NOT EXISTS clauses [79], or recursive SQL queries [62]. Although the coalescing query in this alternatives is relatively shorter than the one that is in the previous alternative, and although it requires fewer accesses to the entire temporal relation, the order of the join operation is higher.

In summary, pure SQL support for temporal coalescing queries require multiple accesses as well as super complex join operation among the original tables, which is far from satisfactory to support temporal database model under current SQL framework.

3.4.2 SSC: A Single Scan Coalescing Algorithm

After a careful study of the exact work of a coalescing operation, we discovered how to conveniently use SQL:2003 standards to support coalescing queries, without any extension to current SQL framework. In this section, we first show a novel algorithm SSC to support coalescing with one single scan of the input tuples without join, then use SQL:2003 to implement SSC.

Without loss of generality, suppose we want to coalesce four tuples with the same time-varying attribute value and different time periods, as in Figure 3.4 (a).

Firstly, we detect all individual timestamps from the input tuples, so in this example, we have eight timestamps, as in Figure 3.4 (b). Notice that timestamp t_4 appears twice.

Next, for each timestamp, we need to keep information of whether it is a *tstart*, or an *TEND* timestamp. We maintain two values to keep the total *TSTART* and *TEND* timestamps we have occurred, with initial value (s,0)/(e,0), and update these two values upon every new timestamp, as in Figure 3.4 (c). For instance, for timestamp t_1, because it is a *TSTART* timestamp, we get (s,1)/(e,0). For t_2 which is again a *TSTART* value, we increment total value of *TSTART* timestamp, and get (s,2)/(e,0). At timestamp t_4 where both *TSTART* and *TEND* occur, we first update its *TSTART* timestamp, then *TEND* timestamp.

Last, we can output all coalesced periods, which are from timestamp t_i to t_j, where t_{i-1} has (s,m)/(e,m), and t_j has (s,n)/(e,n). As in Figure 3.4 (d), there are two coalesced periods: t_1 to t_5, and t_6 to t_7. Intuitively, at timestamp t_i, all previous periods have been output as one coalesced period, and a new coalescable period begins from t_i. At t_j, we have seen equal *TSTART* and *TEND* timestamps, which means t_j should be the *TEND* timestamp of the coalesced period to be output.

With an in-depth analysis of this algorithm, we can find that if all the timestamps are ordered in the input, we can output all the coalesced periods in a single scan of all the input tuples. And in the reality of transaction time databases, all the timestamps are indeed ordered naturally with the passage of transaction time, this guarantees the efficiency of SSC algorithm.

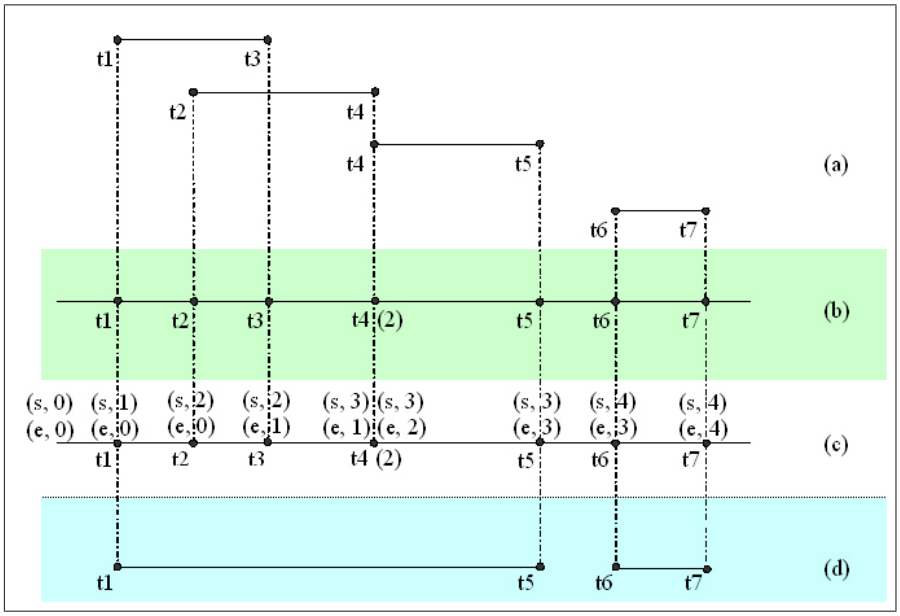

Figure 3.4: SSC: Single Scan Coalescing Algorithm

3.4.3 SQL:2003 Implementation of SSC

With the introduction of SQL:2003 analytics functions, SSC algorithm can be conveniently supported by SQL:2003 statements.

Example 3.17 *SQL:2003 implementation of coalescing query in Example 2.4.*

```
WITH T1 (Start_ts, End_ts, ts, salary) AS (
    SELECT 1, 0, TSTART, SALARY
    FROM EMPLOYEE_HISTORY
    WHERE EMPNO = 1001
    UNION ALL
    SELECT 0, 1, TEND, SALARY
    FROM EMPLOYEE_HISTORY
    WHERE EMPNO = 1001),

T2 (Crt_Total_ts, Prv_Total_ts, ts, Salary) AS (
```

```
    SELECT
        sum (Start_ts) - sum(End_ts)
        OVER (PARTITION BY Salary
        ORDER BY ts, End_ts ROWS UNBOUNDED PRECEDING),
        sum (Start_ts) - sum(End_ts)
        OVER (PARTITION BY Salary
        ORDER BY ts, End_ts
        ROWS BETWEEN 1 PRECEDING AND UNBOUNDED PRECEDING),
        ts, Salary
    FROM T1
    WHERE Crt_Total_ts = 0 OR Prv_Total_ts = 0 )
SELECT Salary,
    max(ts) OVER (PARTITION BY Salary ORDER BY ts ROWS 1 PRECEDING),
    ts
FROM T2 WHERE Crt_Total_ts = 0;
```

In this implementation, the first temporary table *T1* extracts all *TSTART* or *TEND* timestamps from the input tuples, where 1 in *Start_ts* (or *End_ts*) column means it is a *TSTART* (or *TEND*) timestamp. For a certain timestamp value t_i, table *T2* counts the difference between total number of *TSTART* and *TEND* timestamps until t_i (stored as *Crt_Total_ts*), and t_{i-1} (stored as *Prv_Total_ts*). So if *Crt_Total_ts* equals to 0, this means there are equal number of *TSTART* and *TEND* timestamps at t_i, and t_i should be an *TEND* timestamp in one of the coalesced periods. If *Prv_Total_ts* equals to 0, this means there are equal number of *TSTARt* and *TEND* timestamps before t_i, so t_i should be a *TSTART* timestamp in one of the coalesced periods. The final SELECT clause pairs all the result timestamps and output them.

Such an SQL statement can be predefined as a built-in coalescing function, which are transparent to the users. The beauty of this SQL statement is it only

requires one scan of the input tuples, since all the timestamps are already ordered in the reality. With this approach, we can implement all kinds of coalescing functionalities under the current relational database framework, without any complex extension for extra temporal applications.

3.4.4 Generalized SQL:2003 Implementation for Coalescing

The basic SQL:2003 implementation for *SALARY* coalescing query on employee 1001 can be very easily extended to handle all kinds of complex coalescing queries, on different attributes.

- Coalescing on single attribute

  If we remove the condition that employee's *EMPNO* equals to 1001, the SQL:2003 query in Section 3.4.3 is for single attribute coalescing. For another example, if we want to return the history information of the valid periods for each *DEPTNO*, what we need to modify upon the original query is (1) remove the WHERE condition in *T1*, and (2) replace *Salary* with *DEPTNO* for every sub query.

- Coalescing on multiple attributes

  If we want to return the salary history information for every employee, instead of a single employee 1001, this amounts to a coalescing query on two attributes, *EMPNO* and *SALARY*. In this case, what we need to modify upon the original query is (1) remove the WHERE condition in *T1*, (2) also select *EMPNO* in the return of *T1*, and (3), use "PARTITION BY EMPNO, SALARY" to replace "PARTITION BY SALARY" in every sub query.

3.4.5 Support Coalescing with User-Defined Aggregates

Besides using SQL:2003 to support temporal queries in current relational DBMS, another approach that does not require any major extension is using UDAs. User-defined aggregates (UDA), which is an important SQL extension for a lot of complex data mining applications and streaming queries, have been introduced in Oracle's latest version [7]; our Stream Mill system [28] also allows the native definition of UDAs directly using SQL. UDAs can be easily used to support windows, time-series queries, and sequence queries on data streams.

As a result, we can implement a similar single scan algorithm as SSC, directly using SQL-compatible UDAs, and integrate it as system predefined aggregates for users to invoke. Due to space limitation, we only list the pseudo-code as in Algorithm 1.

Such a UDA can be correctly evaluated with the input tuples ordered by the *TSTART* values, which is realistic in transaction databases. The first (TSTART, TEND) input tuple is stored in Temp table (line 1-2). If the incoming tuple intersects with the current period in Temp table, the *TEND* value in the table will be updated to reflect the new end value (line 4-6). If the input tuple does not intersect, we get one result in the Temp table, which needs to be output, and deleted from Temp table, and the new input tuple will be stored into Temp table (line 7-9). We also return the final coalesced period after the last input tuple (line 12).

We will show in next section that this UDA approach beat the traditional pure SQL coalescing queries, although it is not as efficient as the SQL:2003 SSC implementation we discussed in the previous section. But the point is that UDA provides a native SQL support for a lot of advanced queries, such as this classical temporal coalescing query, which otherwise will need external programming

Algorithm 1 UDA Pseudo-code for coalescing query with single scan

```
1: Define table Temp (TSTART, TEND) to store the current coalesced period,
   initially empty;
2: Insert the first tuple's TSTART and TEND value into Temp;
3: for every new input tuple T do
4:    if T.TSTART <= Temp.TEND then
5:       //new tuple coalescable with current period
6:       Update Temp.TEND with T.TEND;
7:    else
8:       //current coalesced period ends, a new coalescing period begins
9:       Output the tuple in Temp, then update Temp with T.TSTART and
         T.TEND;
10:   end if
11: end for
12: Output the tuple in Temp;
```

language to solve SQL's query limitation. Actually, all other complex temporal aggregates - for instance, one returning the history of average salary for all employees - can be efficiently supported with native UDA, which only require one single scan of the input tuples. Due to the space limitation, we omit the details here.

3.4.6 Performance Study

We compare the performance of (i) our proposed SQL:2003 implementation for SSC coalescing query, (ii) user-defined aggregates approach, and (iii) traditional pure SQL implementation with recursive queries. For the experiment, we choose Oracle 10G Release 1 as our DBMS. We executed all of our queries on a personal computer, equipped with a single AMD Athlon XP 1500+ processor at 1.3 GHz, and 512 MB of memory. The operating system we run on is Fedora Core 3 Linux from Red Hat.

The installed version of Oracle Database comes with all the SQL:2003, recursive SQL, and UDA features.

We choose a simulated employee history database as our test data, with the records over 17 years, 20 departments, 4 possible titles. Possible transactions are salary increase, change of titles and departments. The data schema follows that in Figure 3.1. The total transaction database size is 120MB.

First, we test the performance of single-attribute coalescing. We run two queries, coalescing on *deptno*, and coalescing on *title*, respectively. The execution time is shown in Figure 3.5.

Basically, the execution time of our SQL:2003 implementation for SSC is the best of all. We include the pre-compilation time and initialization time of the UDA approach, hence it has longer time than SQL:2003. But actually if we

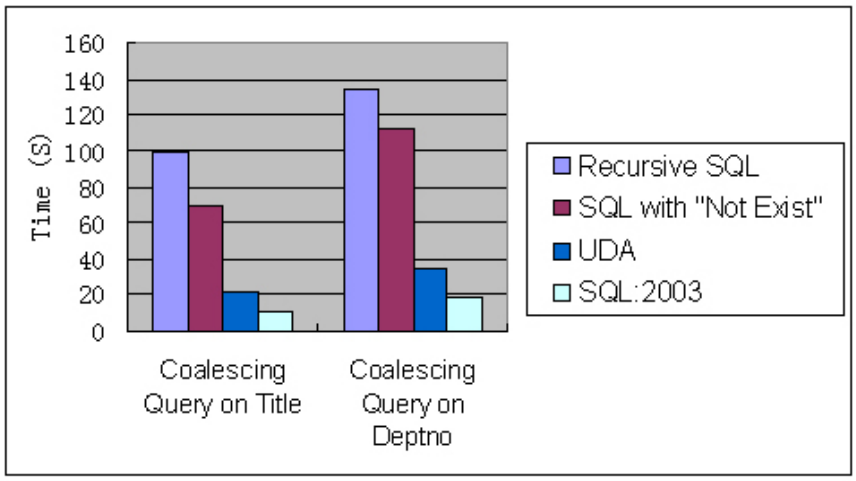

Figure 3.5: Query Performance on Single Attrbute

run the experiments several times, the UDA has very close execution time with SQL:2003, which proves that UDA is also a good choice for coalescing algorithm. The traditional SQL implementation such as recursive SQL or SQL with "Not exist" is much slower than the previous two. But it is still feasible, for single-attribute query.

However, when it comes to two-attribute query, for example, coalescing on *empno* and *title*, or on *empno* and *deptno*, the traditional algorithm takes extremely long time to get the result. We have to test the four queries, on one third of the original transaction database size, and get the result reported in Figure 3.6. The difference ratio is similar with that in Figure 3.5, except that every query has a longer execution time, due to more returns in the output.

We further test the scalability of our SQL:2003 query, with two-attribute coalescing, on 1/4, 1/2, 3/4, and whole data set. Figure 3.7 shows that our algorithm is linear scalable to the increase of data size, which conforms to its single scan feature.

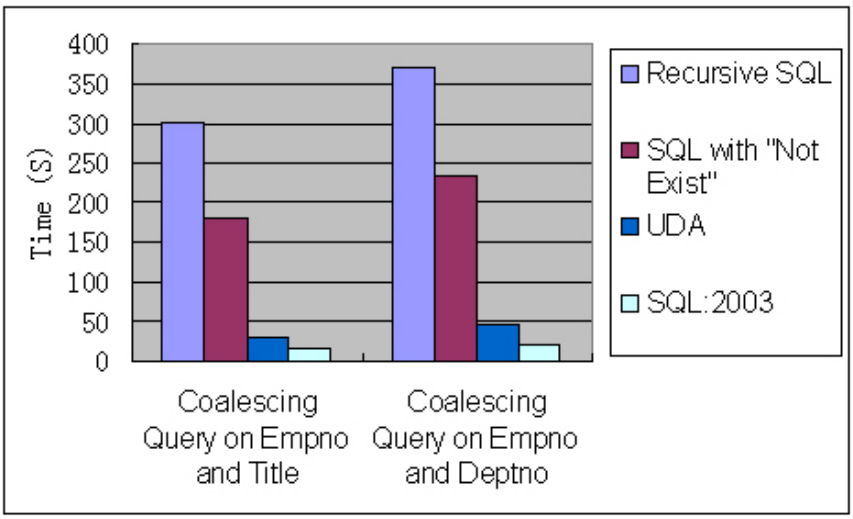

Figure 3.6: Query Performance on Two Attrbutes

3.5 Summary

An important conclusion emerges from the research presented in this Chapter: a unified multi-model support for transaction-time databases can be achieved effectively using a temporally grouped data model. This requires the introduction of new temporal functions and aggregates, but no extension to the current standards. This conclusion holds for all three temporal representations we have studies, namely, XML-based historical view, nested-relations, and OLAP based event tables. We have also developed a unified efficient implementation which relies on well-understood query mapping/optimization techniques, and temporal clustering/indexing techniques at the internal level. Our ArchIS approach is desirable since it provides a low-cost approach to address a wide range of applications. In particular, XML-based views dovetail with web applications, while nested-relations are more natural for object-oriented applications, and the null-filled flat tables are best for traditional database applications, decision-support applications, and event-oriented queries. This last approach provides a simple

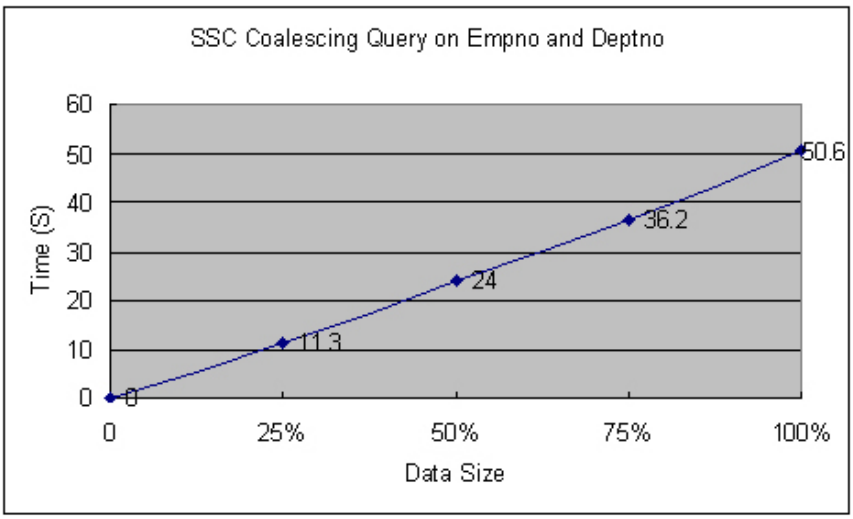

Figure 3.7: Query Scalability of SQL:2003 Implementation of SSC

framework for the presentation of the data, which can require significantly more effort when XML is used. Figure 3.8 summarizes the features of the temporally-grouped schemes proposed, comparing them to the basic ungrouped scheme.

We also show that with SQL:2003 statements or UDAs, we can conveniently support a novel coalescing algorithm, with satisfactory performance. Such an approach makes our system more convincing and worthwhile for further study to be integrated to commercial DBMS vendors.

Scheme / Feature	Ungrouped	XML	Nested Relations	OLAP Tables
External Schema	Flat Tables	XML View	Nested Tables	Null-filled Flat Tables
Temporal Model	Ungrouped	Grouped	Grouped	Grouped
Query Language	SQL	XQuery	SQL:2003	SQL
Temporal Coalescing	Needed Very Often	Seldom Needed	Seldom Needed	Seldom Needed
Event Support	No	No	No	Yes
DBMS Support	ALL	Many	Some	ALL

Figure 3.8: Temporal Scheme Comparison

CHAPTER 4

A Unified System for Relational and XML Data Streams Processing

There is much current interest in processing streaming XML data, using queries expressed with languages such as XPath and XQuery [26, 47, 44, 95]. Meanwhile, a parallel line of research is focusing on the design of data stream management systems (DSMSs) that support continuous SQL queries over relational data streams [17, 31, 49, 63]. The integration of these two approaches is highly desirable because of the same considerations that now drive the seemingly unstoppable push to manage XML documents in traditional DBMSs. This strong drive for closer integration is created by the fact that many stream applications have to deal with both relational data streams and streaming XML data, combine them, and publish results in any combination of the two formats. The technical benefits expected from this integration are also significant and include (i) consolidation of the two competing efforts now spent on system building and marketing and (ii) synergism between the two technologies by combining their respective areas of strength. For instance, research on streaming XML has produced advanced FSA-based techniques for supporting multiple complex queries on structured documents [47, 44, 95]. On the other hand, relational DSMSs are already providing solutions for many problems that remain unsolved in the framework of XML streams. For instance, DSMSs have shown how windows and other synopses are

much needed in continuous queries to overcome the blocking behavior of traditional aggregates and to support efficient queries under limited memory [17, 49]. These constructs are now supported efficiently by all relational DSMSs and they will be very useful on XML streams as well.

In this chapter, we present the design of a unified system that integrates the representation of the two kinds of streams and their processing. Our system supports continuous queries written in SQL, XQuery, and in a combination of the two languages—e.g., XQuery statements can use SQL:2003 OLAP functions and other user-defined functions.

The architecture of our system was chosen after evaluating and discarding alternative designs. An obvious approach consists in building two subsystems—one for XML streams and the other for relational streams—and using pipes to communicate between them while converting between the two types of streams. This approach was discarded in favor of the approach presented in this paper that is preferable because it (i) avoids duplication of implementation efforts, and (ii) achieves tight integration of the data models and query languages. Our approach consists in extending a relational DSMS with efficient support for XML streams and XQuery. For this purpose, we use Stream Mill [28] and its Expressive Stream Language (ESL) that complies with SQL:2003 standards, but also supports efficiently user-defined aggregates because of their proven ability to express very complex queries [27, 61].

Our approach to support XML streams consists in encoding SAX events as relational tuples, and then using ESL to express the wide variety of tasks that integrate relational and XML streams. Queries expressed in XPath or XQuery are thus compiled into equivalent ESL programs that call on user-defined aggregates (UDAs) or system-defined aggregates (SDAs). In the rest of the chapter,

we describe this approach and the many benefits it offers from an application viewpoint. We also show that the various FSA-based optimization techniques proposed for XPath/XQuery [47, 44, 95] can be supported well in our approach, by either UDAs natively defined in ESL, or UDAs defined in procedural programming languages, or SDAs built into the system. In particular, we discuss ESL implementation of YFilter, which supports the parallel processing of multiple XQuery statements, without any extension to current relational query language.

4.1 Relational and XML Data Unification

In this section, we discuss how to create, query, and unify relational and XML data streams. All of these objectives are realized by the Expressive Stream Language (ESL), which is the application language of the Stream Mill system that supports:

- Continuous queries (CQ) on data streams,
- Ad hoc queries on database tables, and
- Spanning applications that combine and compare streaming data with stored data.

To facilitate the learning of the language and its use on spanning applications, ESL is based on SQL and minimizes extensions from its SQL:2003 standards [28, 27].

4.1.1 Relational Data Streams

In the Stream Mill system, each data stream is imported from an external wrapper via the (mandatory) *SOURCE* clause in its *CREATE STREAM* declaration.

For example, consider a hypothetical online web store for book auctions, where both new and used books are traded. Bidders can place bid on books with a certain BookID. Let us assume that bids for used books arrive as the relational data stream declared as follows:

Example 4.1 *Relational data streams definition.*

CREATE STREAM *UsedBookBidStream (BookID* int*,*
BidderID char(10)*, BidPrice* real*, BidTime* timestamp*)*
SOURCE *'port4445'*

Example 4.1 above uses a wrapper (SOURCE 'port4445') that is created automatically by the system for each port used in the program. Rather than using these defaults, users can easily create their own wrappers as described in [28].

New streams can be defined and transduced from existing streams, in a fashion similar to that used to define virtual views in SQL. For instance, to derive a stream consisting of the bids where the bidding price is above 200, we can write:

Example 4.2 *Performing Selection operation on streams.*

CREATE STREAM *expensiveItems* AS
SELECT *BookID, BidderID, BidPrice, BidTime*
FROM *UsedBookBidStream* WHERE *BidPrice* $>$ *200*

4.1.2 XML SAX Event Data Streams

Say that, besides the relational stream of bids on used books, there is another stream of XML messages, which records the bidding information on new books. A sample bid may look like that of Figure 4.1, below:

These streaming XML bids are parsed using the Simple API for XML (SAX).

```
<Bid  BidTime = "2005/02/25T13:24:34">
   <BookID>100001</BookID>
   <BidderID>TC0027</BidderID>
   <BidPrice>65.00</BidPrice>
</Bid>
```

Figure 4.1: A sample XML message for a bid

SAX is a standard interface to parse streaming XML [23], providing a sequential view of an XML document as a stream of events. In Stream Mill, we then represent the SAX stream using a triplet-based format, (*event, name, value*), that we call SAX-3. Thus, the following SAX-3 relational stream is generated, by a SAX based XML parser, from the XML stream of Figure 4.1:

('start', 'Bid', _),

('attr', 'BidTime', '2005/02/25T13:24:34'),

('start', 'BookID', _),

('text', _, '100001'),

('end','BookID', _),

...

('end', 'Bid', _)

Here, 'start' denotes the start-of-element event, and 'end' denotes the end-of-element event. In these two triplets, the *name* column contains the element name, and the *value* column is null. The second triplet shows that each attribute of an element is represented by an 'attr' event with the remaining two columns storing the attribute name and value, respectively. Another special event is 'text', for which the second column is null and the third column contains the actual text.

The following statement defines a SAX-3 event stream in ESL:

Example 4.3 *XML SAX-3 event stream definition.*

CREATE STREAM *SAX-3-Events (event* varchar(10)*,*
name varchar(50)*, value* varchar(50) *)*
SAXSOURCE *'port4448'*

The *SAXSOURCE 'port4448'* clause in this stream definition specifies the port where we have a special wrapper to "wrap" XML SAX events. The wrapper basically takes in the SAX events from the SAX based XML parser and transforms the events into (*event, name, value*) triplet structures.

In addition to this 'vanilla' wrapper, Stream Mill can support more specialized wrappers for more efficient and compressed representation of SAX events. For example, the length of data type *varchar* can be adjusted if the DTD or XML Schema is available. Furthermore, for element like <*BookID*>, which is a leaf element containing only plain text, we can combine the two consecutive SAX-3 event tuples; for instance ('start', 'BookID', _) and ('text', _, '100001') can be merged into tuple: ('start', 'BookID', '100001').

In Stream Mill, we can use arbitrary XQuery FLOWR statements to write continuous queries on XML streams. Such queries take SAX-3 events as input and return SAX-3 events as output. For instance, to derive a stream consisting of the bid XML messages where the bidding price is above 200, we can write:

Example 4.4 *Simple XQuery on a SAX-3 event stream.*

CREATE STREAM *NewSAX-3-Events* AS *(*
FOR *$b* IN *Stream(SAX-3-Events)//Bid*
WHERE xs:real*($b/BidPrice) > 200*
RETURN *($b))*

Queries written in XQuery return SAX-3 event streams, and output wrappers can then be used to produce streaming XML documents from these.

The support for XPath/XQuery statements in ESL will be discussed in Section 4.3, where we also discuss the role of UDAs in the FSA-based parallel processing of multiple XML queries.

4.1.3 From SAX Streams to Relational Streams

SAX-3 event streams are normal relational streams of triplets and thus can be processed using ESL. This allows us to support applications where XML streams and relational streams must be combined. For instance, suppose we need to merge the bid streams of old books and new books. We need to transform the XML-structured new-book bids of Figure 4.1 into the 4-column flat relational format of Example 4.1. However, the bids on new books are first transformed into a stream of SAX-3 events, thus we transform these SAX-3 events to the relational tuples, such as (100001, 'TC0027', 65.00, '2005/02/25T13:24:34'). This transformation in SQL would require the computation four self-joins, which is an expensive operation, of the SAX-3 stream. However, this operation can be easily expressed and efficiently implemented via user-defined aggregates (UDAs) supported in ESL, as shown in Example 4.5. (The actual definition of *Flatten()* is given later.)

Example 4.5 *Creation of relational data streams out of XML SAX-3 event streams.*

```
CREATE STREAM NewBookBidStream AS (
    SELECT Flatten(event, name, value)
    FROM SAX-3-Events)
```

To implement the functionality of Flatten() UDA, what we need is to detect an attribute event for *BidTime*, and three consecutive start of element events for *BookID*, *BidderID*, and *BidPrice*. Then, we flatten their values into one tuple, which follows the relational schema of Example 4.1, whereas all other SAX-3 events are discarded. (The UDA assumes that the SAX stream wrapper already wraps the text value of a leaf element into its *value* field, thus we get SAX-3 events such as ('start', 'BookID', '100001')).

Example 4.6 defines such a UDA to flatten SAX-3 events into relational tuples. In this UDA, a local table *Temp* is defined, where the BookID, BidderID, BidPrice and BidTime values are kept.

Example 4.6 *Definition of a UDA to flatten SAX-3 event streams into relational streams.*

```
AGGREGATE Flatten (event varchar(10), name varchar(50), value varchar(50))
: (int, char(10), real, timestamp) {
TABLE Temp(BookID int, BidderID char(10), BidPrice real, BidTime timestamp);
INITIALIZE: { }
ITERATE: {
    INSERT INTO Temp VALUES (Null, Null, Null, CAST(value, timestamp))
        WHERE event = 'attr' AND name = 'BidTime';
    UPDATE Temp SET BookID = CAST(value, int)
        WHERE event = 'start' AND name = 'BookID';
    UPDATE Temp SET BidderID = value
        WHERE event = 'start' AND name = 'BidderID';
    UPDATE Temp
        SET BidPrice = CAST(value, real)
        WHERE event = 'start' AND name = 'BidPrice';
    INSERT INTO RETURN
        SELECT * FROM Temp
        WHERE BookID != NULL AND BidderID != NULL
```

```
        AND BidPrice != NULL AND BidTime != NULL;
    DELETE FROM Temp WHERE SQLCODE = 0; }
TERMINATE: { } }
```

In the ITERATE block of this UDA, a new tuple is inserted in the *Temp* table when a *BidTime* attribute event is read, and its value (casted to the correct data type) is stored in the *BidTime* field. When the input is the start of element for either *BookID*, or *BidderID*, or *BidPrice*, the corresponding field of the tuple is updated. The tuple in *Temp* will be returned once each field has been filled with a value. Once a tuple in *Temp* has been returned (an event that sets the system variable SQLCODE to 0), it is no longer needed and it is deleted.

The TERMINATE block is empty in this example, since satisfied tuples have already been returned in ITERATE. Therefore, this UDA is non-blocking; indeed it serves as a stream transducer, where streams are piped in and piped out.

Similar UDAs can be defined to flatten any number of elements. For example, if we are only interested in bid price and time, we can write a UDA which generates (BidPrice, BidTime) pairs and feeds into another 2-field relational stream.

An immediate benefit of this flattening is that the streams for used books, *UsedBookBidStream*, and new books, *NewBookBidStream*, can now be merged using the union operator to support continuous queries on bids for both new and used books.

Union is a very important operation on data streams, and can often be used to avoid join operations that might require unbounded memory in extreme cases, or return approximate results by finite window sizes. For instance, suppose that we have a relational stream *CloseStream(BookID, CloseTime)* describing closing auctions. Since we have *UsedBookBidStream*, defined in Example 4.1, for used books and *NewBookBidStream*, defined in Example 4.5, for new books, we can

now get the closing price of each book as soon as its auction closes. The following example illustrates our approach.

Example 4.7 *Return the winning bid's price when a certain auction is closed.*

```
CREATE STREAM BidStream (BookID, BidderID, BidPrice, BidTime) AS (
    SELECT BookID, BidderID, BidPrice, BidTime
    FROM UsedBookBidStream
    UNION
    SELECT BookID, BidderID, BidPrice, BidTime
    FROM NewBookBidStream );

CREATE STREAM AuctionBehavior (behavior, id, price, time) AS (
    SELECT 'close', BookID, Null, CloseTime
    FROM CloseStream
    UNION
    SELECT 'bid', BookID, BidPrice, BidTime
    FROM BidStream;

SELECT PriceReport(behavior, id, price, time)
FROM AuctionBehavior
```

In this example, bid streams for both used and new books are first unioned into one single stream *BidStream*, which is then unioned with *CloseStream* into a new stream *AuctionBehavior*, ordered by its *time* value. The resulting stream is finally passed to the UDA *PriceReport*, which basically returns the closing price of an auction upon reading the "close" event for the auction (see Section 4.2.3 for details).

Conversely, given a certain mapping schema, we can also transform relational data streams into XML SAX-3 streams by using simple UDAs. This makes

it possible to apply to relational data streams, the FSA-based techniques for processing multiple XPath/XQuery statements in parallel that will be discussed in Sections 4.3.

4.2 Query Language Integration

In Section 4.1.3, we outlined how to use ESL to generate relational streams from XML streams, and vice versa. But, in addition to integrating these two types of data streams, we also want to achieve the cooperation of their respective query languages. In particular, we would like to integrate the two languages, making XQuery capable of importing SQL queries, to achieve advanced continuous queries on the unified data streams.

4.2.1 Streaming Queries Using UDAs defined in SQL

There are many research projects that focus on extending SQL to handle new class of applications in data streams. Indeed, applications that span both data streams and static data provide a strong reason for using SQL for querying such data [31]. Unfortunately, SQL is facing severe limitations in this new role — particularly with respect to query power and extensibility. A main objective of the Stream Mill system is to overcome these limitations, and thus achieve a much broader range of usability and effectiveness. Stream Mill supports Expressive Stream Language (ESL), as we have discussed in last section. Native definition of UDAs is the kernel of ESL. It empowers ESL to handle more complex streaming queries via a small extension of SQL. The idea comes from ATLaS [92], which was intern borrowed from SQL-3. Let us use the following very simple example to illustrate the basic idea, which comes from [92]:

Example 4.8 *Definition of UDA to calculate the online avg*

```
1 AGGREGATE online_avg(next REAL) : REAL {
2 TABLE state(sum REAL, count REAL);
3 INITIALIZE: {
4     INSERT INTO state VALUES (sum, 1); }
5 ITERATE: {
6     UPDATE state SET sum = sum + next, count = count + 1;
7     INSERT INTO RETURN SELECT sum/count FROM state; } }
8 TERMINATE: { }
```

Example 4.8 defines an aggregate equivalent to the online AVG aggregate. Line 2 declares a local table, **state**, where the sum and count of values processed so far, are kept. Furthermore, while in this particular example, **state** contains only one tuple, it is in fact a table that can be queried and updated using SQL statements and can contain any number of tuples. Here, INITIALIZE block inserts the values taken from the input stream into the **state** and sets the *count* to 1. The ITERATE block updates the tuple in **state** by adding the new input value to the *sum* and 1 to the *count*. It also returns the ratio between the sum and the count as the final result of the computation by the "INSERT INTO RETURN" statement in line 7. The TERMINATE block is executed right after all the input tuples have been processed, where we do not need to do anything in this case. The definition of all three statement blocks, INITIALIZE, ITERATE, and TERMINATE, in one procedure allows supporting the declaration of their shared tables (the **state** table in this example).

This UDA is actually a non-blocking UDA, which is clearly identified by the fact that its TERMINATE block is empty. This generic aggregate definition mechanism allows tremendous flexibility and power. In fact, ESL with UDAs

is Turing-complete and allows natively specifying complex mining functions on data streams. Advanced applications of UDAs are shown in [92].

4.2.2 Import SQL UDAs into XQuery

Incompleteness of XQuery on Streaming Data. The native definition of user-defined functions allows XQuery to achieve Turing-completeness [60]. Unfortunately, the function-definition mechanism of XQuery is blocking and thus not suitable for streaming data. This can be illustrated by the following example showing the definition and invocation of a count-like aggregate:

Example 4.9 *Return the total number of bids.*

```
DECLARE FUNCTION mycount($bids as xs:AnyType) AS xs:integer {
    IF (empty($bids)) THEN 0
    ELSE mycount(subsequence($bids,2))+1 }

LET $a IN STREAM("AllBids.xml")/Bids/Bid
RETURN mycount($a)
```

The sequence (i.e., the list) of all bids is given to the XQuery function, that then applies the count function to its tail (i.e., the subsequence starting from the second element). As shown in this example, XQuery functions assume that the whole sequence is present and materialized before the computation is started. This computation model is blocking and will not work when we have an infinite stream of records. Namely, the native extensibility mechanisms provided by the current XQuery standards cannot be used to define online aggregates in a streaming way.

Curing the Incompleteness of XQuery on Streaming Data. Therefore, the superior expressive power of XQuery evaporates when processing streaming data. We will take a natural approach of keeping the FLOWR constructs intact and explore mechanisms to introduce non-blocking user-defined functions and aggregates in XQuery. Our solution exploits the fact that XQuery can accept functions defined in external languages—including nonblocking functions. In other systems, the functions to be imported into XQuery would be written in C++ or Java; but the Stream Mill system closely integrates XQuery and ESL and thus allows the importation of SQL:2003 analytical functions as well as window aggregates directly from ESL. This produces a significant simplification for the user and the system alike.

For example, suppose we need to calculate the online avg of a stream of XML elements input. The XQuery and imported function will look like the follows.

Example 4.10 *Return the online avg of bids with XQuery.*

```
DECLARE FUNCTION myavg($bids as xs:AnyType) AS xs:double {
    SELECT online_avg (T.p)(empty($bids)) THEN 0
    FROM Stream($bids) AS T(p) }

FOR $a IN STREAM("AllBids.xml")/Bids/Bid/BidPrice
RETURN myavg($a)
```

Basically, the main body of XQuery is kept intact with current FLWOR standard. The novel part lies in the imported myavg() function, which actually uses the online_avg() UDA we defined in last section. In this way, the function will return a new avg value every time when it sees a new BidPrice element input.

Obviously, more complex UDAs can be defined and imported for more advanced queries on integrated data streams, as we discuss in next section.

4.2.3 Advanced Queries on Integrated Data Streams

We also supports advanced sequence and time-series queries that represent another important application area for data streams.

Example 4.11 *Continuously return all the patterns of two consecutive bids that raise the previous bid by at least 50%.*

```
SELECT BookID, IncreaseDetect(BidPrice)
FROM BidStream GROUP BY BookID;

AGGREGATE IncreaseDetect (BidPrice real) : real {
    TABLE temp (price real, rank int);
    INITIALIZE: {
        INSERT INTO temp VALUES (BidPrice, 1), (BidPrice, 2); }
    ITERATE: {
        INSERT INTO temp VALUES (BidPrice, 3);
        INSERT INTO RETURN
            SELECT t1.price, t2.price, t3.price
            FROM temp t1, temp t2, temp t3
            WHERE t1.rank = 1 AND t2.rank = 2 AND t3.rank = 3
            AND t1.price * 1.5 <= t2.price AND t2.price * 1.5 <= t3.price;
        DELETE FROM temp t WHERE t.rank = 1;
        UPDATE temp t SET t.rank = t.rank -1; }
    Terminate: { } }
```

In this example, the *BidPrice* is extracted from the original *BidStream* and passed to the UDA *IncreaseDetect*, which detects price increases. A temporary table *temp* holds the last three bids; when two consecutive bid prices exceed the previ-

ous bid by 1.5, the UDA returns the content of *temp*, i.e., the sequence of three successive price increase by 50% or more. This example illustrates the ability of UDAs to support state-based computations. Without UDAs, or similar state-based operators, detecting a sequence of n events normally takes $n-1$ self-joins, which is a very complicated operation on data streams.

Let us now return to Example 4.7 in Section 4.1.3, where we pass the combined stream *AuctionBehavior* to a PriceReport() UDA to determine the winning price. The definition of this UDA is given below:

Example 4.12 *Return the winning bid's price when a certain auction is closed (UDA which is called in Example 4.7).*

```
AGGREGATE PriceReport (behavior varchar, id int, price real, time timestamp)
: (int, real, timestamp) {
    TABLE temp (item int, currentprice real);
    INITIALIZE: {
    ITERATE: { }
        UPDATE temp
            SET currentprice = price
            WHERE behavior = 'bid' and item = id;
        INSERT INTO temp VALUES (id, price)
            WHERE behavior = 'bid' AND SQLCODE != 0;
        INSERT INTO RETURN
            SELECT t.item, t.currentprice, time
            FROM temp t
            WHERE behavior = 'close' AND t.item = id;
        DELETE FROM temp t
            WHERE SQLCODE = 0 AND t.item = id; }
    TERMINATE: { } }
```

In Example 4.12, the first two SQL statements in ITERATE are used to update current highest bid price for each item. The last two SQL statements in ITERATE

are used to output the winning price when the auction of a certain item closes. UDAs such as those in Examples 4.11 and 4.12 are non-blocking, and can be used directly in ESL, or in XQuery functions as discussed in next Chapter.

4.3 Efficient Support for Multi-Query Processing

In Sections 4.1.3, we discussed how to transform XML streams into relational streams, where the input XML messages are flattened into relational tuples, as shown in Examples 4.5 and 4.6. Here, the schema for the input XML, consisting of bids as those of Figure 4.1, is rather simple and regular, and our queries are also simple. In general, we must deal with complex XML structures and arbitrary queries.

```
<Auction>
 <Book BookID = "100001">
  <Title>A Complete Guide to DB2</Title>
  <Author>Don Chamberlin</Author>
  <Content>
   <Chapter>
    <Title>Introduction</Title>
   </Chapter>...
  </Content>
 </Book>
</Auction>
```

Figure 4.2: A sample XML message for book information

For instance, suppose a user is interested in writing queries to extract 'books which have a $<Title>$ descendant element containing keyword "DB2"', from XML messages such as that of Figure 4.2. The user would normally prefer to express

his/her query in XQuery/XPath, rather than writing a flatten UDA such as that of Example 4.6. Indeed, Stream Mill allows users to write continuous queries directly using XQuery/XPath. These queries are then supported using UDAs that implement the FSA-based approach of YFilter [95] for performance and parallel execution.

4.3.1 UDA Simulation of Basic FSA

We will begin with a very simple example to explain the idea of FSA-based XPath processing as in [95], using UDAs written in SQL. For better performance these aggregates have eventually been implemented as system-defined aggregates (SDAs), that make full use of Stream Mill's internal optimization techniques.

Suppose that an XPath query Q0, */Auction/Book/Author*, is issued by the user to filter the input documents. A simple FSA can be built to simulate the processing of Q0 based on the input SAX events, as in Figure 4.3 (a), where the name on every edge represents the triggering element name between two states.

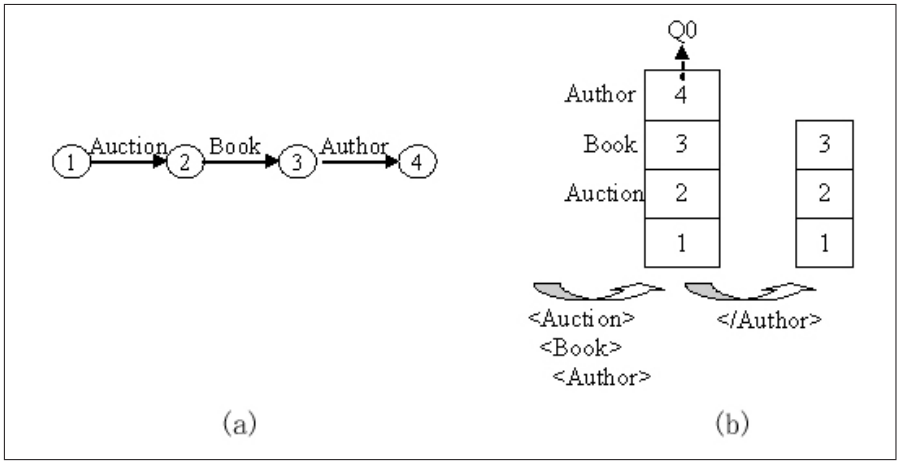

Figure 4.3: FSA for Q0: /Auction/Book/Author

A runtime stack is maintained to keep the active states while we read in

SAX events. The transition between states is triggered by the start event of an element, whereby new active states are pushed on top of the stack; similarly, the end event will perform backtracking by popping out the active states from the top of the stack. Every time we reach the accept state, we find an answer to the XPath and can return it on the fly. The runtime stack based on the XML input of Figure 4.2 is shown in Figure 4.3 (b).

Based on the non-blocking calculation feature of relational UDAs, as well as its Turing-complete expressive power, we aim at building a simulation for the execution of FSA, where we can bridge the different computational styles of FSA and SQL, and share them under the same relational data model.

We use (i) a **Transition** table to store the transition graph, such as that of Figure 4.3 (a), and (ii) a **State** table to store the states of FSA (in the *SType* column, 'i' means initial state, 'm' means middle state, and 'a' means accept state). The table schema and sample content are as follows:

State Table

SID	SType	QID
1	i	-
2	m	-
3	m	-
4	a	Q0

Transition Table

SrcID	symbol	DestID
1	Auction	2
2	Book	3
3	Author	4

Observe that in **State** table above, only the accept state has a *QID* value giving the ID for the satisfied query when the FSA reaches that state. Here, again, we will generate a SAX-3 event stream as a result of incoming XML messages.

Transition and backtracking are done in the ITERATE state for every input event in *SAX-3* stream. The basic UDA definition is shown in Example 4.13;

observe that the code in every state can be implemented in just one or two SQL statements.

Example 4.13 *UDA simulation for simple XPath queries.*

```
Table State (SID int, SType char(10), QID char(2));
Table Transition (srcID int, symbol char(50), destID int);

Aggregate FSA (event char(10), name char(50), value char(10)): (QID char(2)) {
TABLE RuntimeStack (level int, SID int);
TABLE StackTop (level int); /*Used to decide the stack top*/
INITIALIZE: {
    /* Push initial state in State table into RuntimeStack table*/ }
ITERATE: {
    /*start of element handler*/
        /* If (the top state in RuntimeStack is 's1',
                AND input event is 'start',
                AND input event name is 'e'
                AND entry (s1, e, s2) is in Transition table)
            Push s2 on top of RuntimeStack */
    /*decide accept state*/
        /* If (the top state in RuntimeStack is an "accept" state in State table)
            Return the QID of that state in State table */
    /*end of element handler*/
        /* If (the input event name is 'end')
            Delete the top state in RuntimeStack table */}
TERMINATE: { } }
```

This UDA only returns the IDs of the queries that are satisfied, but this basic scheme can be extended to output SAX-3 events of matched elements.

Notice that TERMINATE state is empty in the previous UDA, and satisfied queries are returned on the fly. Therefore, this UDA is actually a non-blocking

UDA, which pipelines the input and the output streams. This is exactly what we need for streaming XML processing.

4.3.2 Multiple Complex XPath/XQueries

The previous section described how to filter XML documents for a simple XPath query (/Auction/Book/Author), however several extensions are required to enable parallel processing of more complex XPaths as in YFilter.

4.3.2.1 Multiple XPath Queries with Wild-card '*' and Descendent Axis '//'

As discussed previously, the XPath statement /Auction/Book/Author is translated into the FSA of Figure 4.3(a), where '/Auction' corresponds to the transition from state 1 to state 2. The presence of wild cards in our query statements do not change this overall translation scheme by much; for instance, if the above query is changed to /*/Book/Author, then the transition from state 1 to state 2 in Figure 4.3(a) will have '*' as its label. However, as pointed out in [95], the use of the descendant axis brings additional complications to this translation. For instance, if the above query is changed to //Auction/Book/Author, then an extra state, state 0, is needed, along with a transition from state 0 to state 1 with label ϵ, and a transition from state 1 to itself with label '*'. Furthermore, in this case, state 0 will become the start state of the FSA. Thus, integration of these advanced constructs requires some extensions to the FSA translation process. This is also illustrated by the example shown in Figure 4.4 that gives the FSA for XPath /Auction/*//Title.

The ability of supporting parallel processing of a large number of queries represents an important technology developed as a result of several research efforts

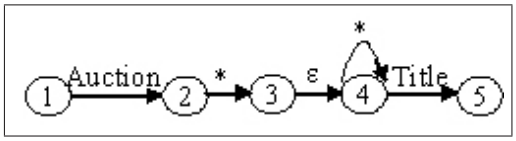

Figure 4.4: FSA for */Auction/*//Title*

on streaming XML data. In particular, YFilter [95] presents a technique for combining FSA that solves this problem efficiently, by assuring that the prefixes of the different XPath statements are combined and shared as much as possible. As shown in [61], SQL extended with UDAs is Turing Complete, and can thus easily support UDA that represents such combined FSA. We use the Transition table of Example 4.13 to record the transitions of the combined FSA. This Transition table can be built incrementally as new queries are added/deleted. The addition of the wild-card, descendant axis, and parallel processing of multiple XPaths require small extensions to the UDA of Example 4.13; the details are omitted here due to space constraints. This combined-FSA UDA presents many opportunities for optimization, for instance use of indexes and memory tables. Furthermore, the UDA can also be supported as a system built-in function, in which case it can be written in an external programming language and can take advantage of specialized optimization techniques.

4.3.2.2 XPath/XQueries with Branch Queries

Besides the simple linear XPaths we have discussed above, our XPath statements can actually contain structure-based or value-based predicates. For example, /Books/Book[Author]/Title will return book titles only for those books that contain author information. This XPath expression can be divided into the two subexpressions Q1: /Books/Book/Author, and Q2: /Books/Book/Title. Before a result can be returned to the output, the sub-results produced by these two

subexpressions must be joined on their shared *Book* node.

Building on the techniques presented above, we can now extend our UDAs to process complex XPath and XQuery statements by decomposing each query into separate XPaths. But in this case, in addition to returning the satisfied query ids and the corresponding SAX-3 events, we also need to return, for each matched element, an internal element id that is needed later for joins. Such unique id can be easily maintained by creating an id table with just one tuple, and incrementing it every time a new start-of-element event is detected. For instance, say that the queries Q1 and Q2 discussed above are issued on the following XML input, where the number associated with each start element represents the internal element id:

$< Books >^1$

$< Book >^2$

$< Title >^3$ A Complete Guide to DB2 $< /Title >$

$< Author >^4$ Don Chamberlin $< /Author >$

$< Content >^5 ... < /Content >$

$< /Book >$

$< Book >^6$

$< Title >^7$ Advanced Database Systems $< /Title >$

$< /Book >$

$< /Books >$

The result set for Q1 (/Auction/Book/Author) contains only one match, namely, {1-2-4}, where the numbers represent the satisfied element ids. Similarly, the result set for Q2 (/Auction/Book/Title) is {1-2-3, 1-6-7}. However, since the two XPaths need to be joined on their second element, 1-2-4 and 1-2-3 match and 1-2-3 is returned as output, but there is no match for 1-6-7, which thus produces no output.

Our basic UDA can be easily extended to cope with this join requirement, and return all the satisfied element ids for each XPath, as well as the SAX-3 events for the output. All those element ids are then passed to another UDA to perform the join operations and output the SAX-3 events that satisfy the query. Likewise, XQuery statements are first mapped into several XPath statements joined on some common nodes, and then processed with techniques similar to those discussed above.

4.3.3 FSA Application to Multiple SQL Queries

At the end of Section 4.1.3, we mentioned that relational streams can be transformed to XML SAX-3 streams, for a potential benefit of efficient parallel processing of multiple SQL queries. After we explained the FSA simulation earlier in this section, it is natural to apply these parallel techniques for multiple XQuery processing to SQL processing. What we need to do is to define a mapping scheme and transform relational streams into XML SAX-3 streams accordingly. Then translate every SQL query into equivalent XQuery following the mapping schema. All the rest is calling UDA-based or external-function-based FSA execution on the streaming input, and answers will be pipelined to the output for different SQL queries.

4.4 System Implementation and Performance

4.4.1 Overall System Architecture

The Stream Mill system unifies relational data streams and XML streams using the architecture outlined in Figure 4.5. Stream Mill supports the conversion of XML SAX-3 events to flat relational tuples and vice versa, and this paves the

way to a closer cooperation of relational and XML query languages, resulting in greater power and flexibility.

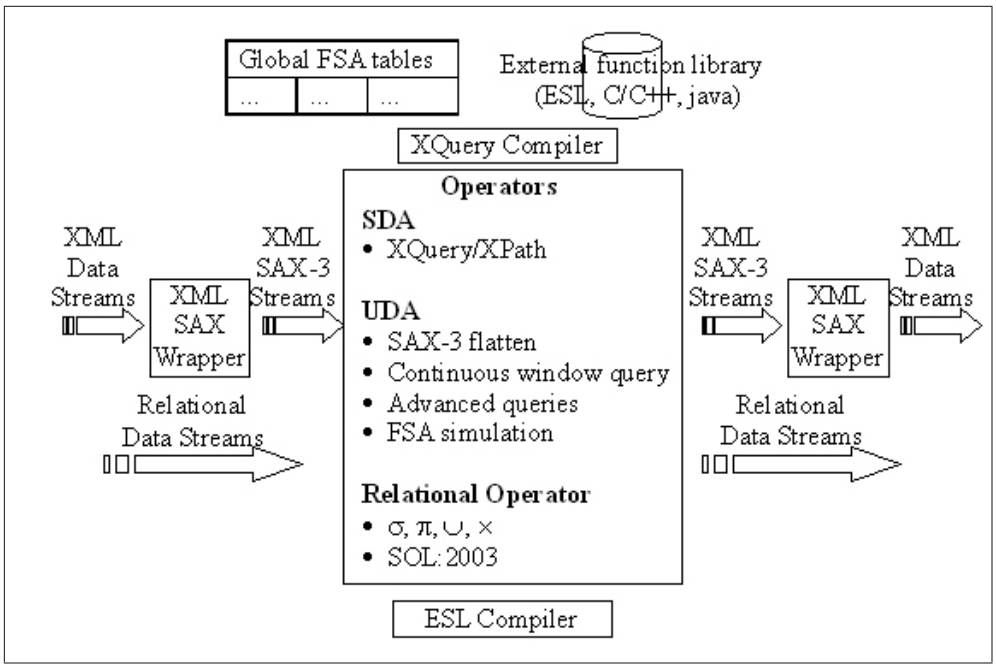

Figure 4.5: Architecture for unifying XML data streams and Relational data streams

Stream Mill supports normal selection, projection, union, and special join operations on relational data streams. The system also integrates SQL:2003 OLAP standards applicable to data streams. The most distinctive feature of the Stream Mill system is its support for native UDAs that allow advanced continuous queries on relational streams. These functionalities can now be used on flattened SAX-3 streams.

To achieve the full integration of relational and XML streams and their query languages, the Stream Mill system also supports XQuery so that users can express queries on streaming XML documents by using standard XQuery. However, XQuery currently fails to provide good support for analytics, data mining, and

many other functions that are now available in SQL:2003, or in ESL using our extended UDAs [27]. Therefore, we enable XQuery to call ESL UDAs and allow it to take full advantage of these powerful facilities.

Given an XQuery statement, Stream Mill first decomposes it into several XPaths; these XPaths are added to the parallel XPath processor, which is a UDA similar to that of Example 4.13 and generate the content in global FSA tables. The results of these XPaths queries are joined to determine the results of the original XQuery. Furthermore, specialized external functions and UDAs can be called from XQuery, and the compiler supports dynamic incorporation of such functions, and the insertion or deletion of queries that are processed in parallel by Yfilter UDAs.

4.4.2 Performance Study

ESL is the SQL-based application language for Stream Mill system, a DSMS thta provides full support for ESL queries via functions such as compilation, optimization, query scheduling, load balancing, in-memory tables, hash-based indexes, R-tree based indexes, and performance monitors [28].

Three implementation approaches were explored for the parallel processing of XPath statements. The first option is to use ESL to define UDAs that simulate the Yfilter FSA. As a second alternative, the same UDAs can be written in programming languages such as C/C++, to achieve better performance. Thirdly, system-defined aggregates (SDAs) can be built to support the Yfilter FSA and achieve optimal performance and scalability. In our experiments, we measured the performance obtained with the second approach, i.e., UDAs written in an external PL, and compared it to that of the YFilter demo system [42]. In terms of performance, this a middle-road solution that achieves better performance and

scalability than ESL-coded UDAs, but it not as good as that expected from SDAs. The results of our experiments are reported in Figures 6 and 7, where FSA_ESL_ext denotes the C++ coded UDAs.

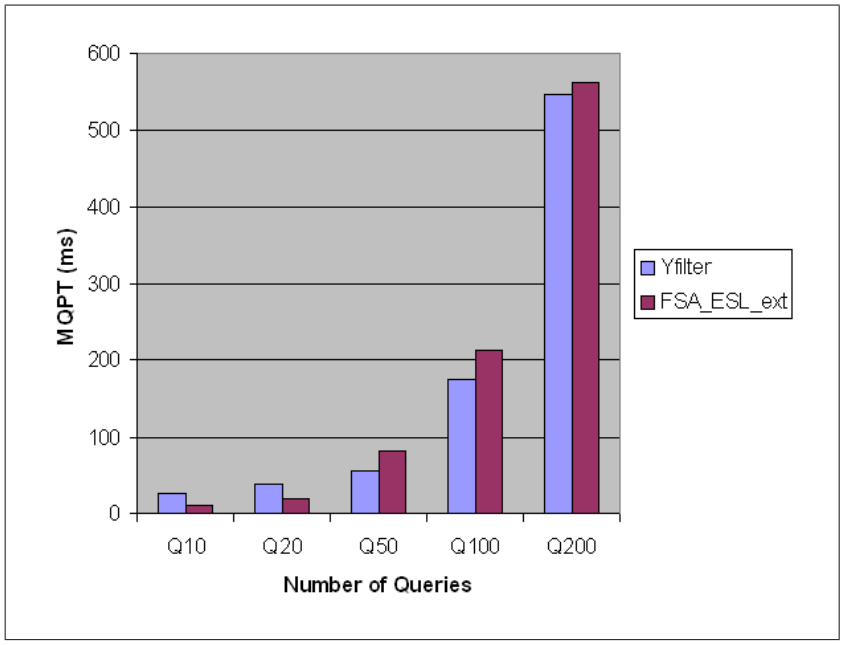

Figure 4.6: Scalability Test Results

All of the experiments reported here were performed on a P4 2.4GHz processor with 1GB memory running JVM 1.4.1 on Linux Red Hat 8.0 machine. As discussed in [95], we use multi query processing time (MQPT), which includes the filtering time but not the document parsing time, as the measure of the efficiency. We perform different experiments to test the scalability and performance for different types of queries. Our experiments suggest that, on the average, the performance of our C++ coded UDAs is comparable to that of the original YFilter, although it can be better or worse for a particular types of queries.

Scalability Test In this test we check the performance of the systems for increasing number of queries. Figure 4.6 shows filtering times for increasing

number of queries: 10, 20, 50, 100, and 200. The filtering times for YFilter and FSA_ESL_ext increase at almost similar rate. This shows that both systems are equally scalable.

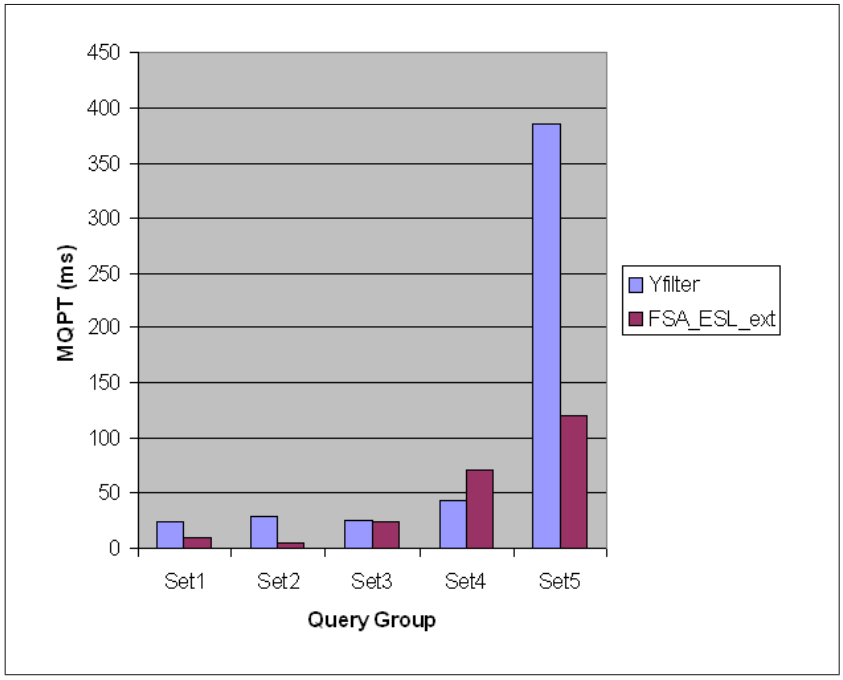

Figure 4.7: Effect of Different Types of Queries

Effect of Different Types of Queries Next, we check the effect of different types of queries. In this experiment, we take five different groups of queries, each group consisting of a set of 50 queries as follows:

Set1: simple queries that do not contain a wild-card characters or a '//' descendant axis

Set2: queries with a 0.2 probability of wild-cards

Set3: queries with a 0.2 probability of '//' descendant axes

Set4: queries with a 0.15 probability of wild-card and a 0.5 probability of '//' descendant axes,

Set5: queries with a 0.4 probability of having wild-cards combined with '//' descendant axes.

Figure 4.7 shows the MQPTs for the respective implementations. FSA_ESL_ext has high MQPT for set4, but YFilter has much higher MQPT for set5. The experiments suggest that, on the average, the UDA-based implementation in the Stream Mill system is comparable to the specialized implementation of YFilter as far as filtering times and scalability are concerned.

We have also implemented our FSA simulator of YFilter as SDAs using C++ and integrated them in Stream Mill. Although this work is still in progress and various optimization improvements remain to be added, our preliminary results show that the performance of our SDAs is always better than that of YFilter implemented in Java—from 2 times to 3.5 times faster, depending on different query groups. This difference is what one would normally expect for respective implementations in Java and C++. In conclusion, building an implementation of XML and XQuery in Stream Mill produces performance results comparable to those obtainable by building a complete new system from scratch, while greatly reducing the effort spent in design and development.

CHAPTER 5

OLAP Query Support and XML

XML has been universally accepted as the data exchange and publishing format among business applications, in both document-centric and transaction-centric domains. XQuery is a widely used standard query language for XML data; it follows the FLWOR navigation mechanism on hierarchical data, and supports built-in and user-defined functions for extensibility.

Online Analytical Processing (OLAP) is a valuable tool for analyzing trends in business information, which has been extensively used in decision support systems. OLAP is well supported by relational database vendors and SQL:2003 standards through an array of aggregation functions and multi dimensional operators, such as GROUP BY, ROLLUP, CUBE, etc.

XML's wide-spread use in different business areas requires that the query language, XQuery, supports functionalities for OLAP applications on XML data model, as SQL does for relational data model. However, the lack of explicit GROUP BY constructs in XQuery makes it inconvenient and inefficient to express simple groups in the hierarchical data model, not to mention more complex grouping sets on multiple dimensions. A lot of open issues exist in the research of designing a simple query language expression, an efficient query plan, and a meaningful structured output for OLAP queries, on XML's unique hierarchical data.

The work presented in [57, 70] proposes supporting GROUP BY constructs similar to SQL, in current XQuery standard. Such approaches do not provide satisfactory solutions to the OLAP query problem due to several reasons. First, to support complex grouping sets, these approaches require writing native XQuery user-defined functions, which are either hard to write or difficult to optimize. Second, the new grouping constructs introduce a lot of modifications to current evaluation model of XQuery, which is problematic for current native XML database vendors. Third, specific query plan and optimizations need to be developed for these queries due to the grouping constructs, thus relational data model's mature indexing techniques for OLAP queries cannot be reused. Although the approach discussed in [57] has been implemented in System RX [57], which is a specialized system, it is far from being integrated in the W3C standard and will further require a lot of changes in the current native XML systems and relational database systems' XML extension.

Another approach would be to use SQL/XML query functions which have already been supported in commercial relational database products such as DB2 [2] and Oracle [5]. XML-related functions such as XMLTable, XMLQuery and XMLExists can be used to extract the XML values from XML typed data. Then the analytics can be applied on the SQL side. Such approach integrates two separate steps, first constructing a relational-like view out of XML data, and then applying OLAP on the relational view. Integrating these two pieces of work together could provide more efficient solution from implementation's point of view, and more user friendly interface from users' point of view.

In this chapter, we introduce a third approach to enable OLAP queries in XQuery. This is achieved with minimum extensions to current FLOWR standard, using the XQuery function import idea discussed in Chapter 4. These

minimal extensions provide significant advantages in user-friendliness, expressive power, and query execution techniques. The basic idea is to allow XQuery to import external functions and aggregates, which are essentially SQL/XML-defined functions and aggregates. All the grouping dimensions and aggregation calculations are easily expressed in the external functions and aggregates, which simply reuse SQL's OLAP grouping constructs. Both value-based OLAP queries, such as those in SQL, and XML's structure-based OLAP queries can be supported efficiently in our framework. Furthermore, these OLAP queries can return specific hierarchical results by using system built-in publishing functions written in SQL/XML.

5.1 OLAP Extensions for XML

We first provide an example to point out some current issues in XML's analytical processing. We briefly discuss the current approaches and show their shortcomings, which motivate our effort to provide an alternative solution in this paper.

We base our use cases on a hypothetical online web store for book purchasing, where used books can be posted online for sale. A sample content of such a sale message is shown in Figure 5.1, where book and sale properties are represented in XML elements.

Namely, every book has one title, one or more authors, one publisher, one category description, one content description, a price, and the time when the sale is posted. Random structured data can be organized under the "category" element. The content description may contain several sections, each of which has an introduction and possibly further sections in it. All such book sale information is concatenated under a root element "books". A typical OLAP query on such

```
<books>
 <book>
  <title>...</title>
  <author>...</author> <author>...</author>
  <publisher>...</publisher>
  <category>
    <software> <db><concurrency/></db> </software>
    <distributed/>
  </category>
  <content>
    <section>
      <introduction>...</introduction>
      <section><introduction>...</introduction></section>
    </section>...
    <section>...</section>
  </content>
  <price>...</price>
  <posttime>...</posttime>
 </book>
 <book>...</book>
 ...
</books>
```

Figure 5.1: book.xml: A sample book sales XML document

document would be as Example 5.1, where four separate XQuery statements need to be written according to current standards.

Example 5.1 *(Current XQuery Implementation) Return the average price, (1) overall, (2) by author, (3) by publisher, and (4) by author-publisher combination.*

LET *$b := doc("books.xml")//book/price*
RETURN {*avg($b)*}

FOR *$a* IN distinct-values *(doc("books.xml")//book/author)*

LET *$b := doc("books.xml")//book[author = $a]/price*

RETURN {*$a, avg($b)*}

FOR *$a* IN distinct-values *(doc("books.xml")//book/publisher)*

LET *$b := doc("books.xml")//book[publisher = $a]/price*

RETURN {*$a, avg($b)*}

FOR *$a1* IN distinct-values *(doc("books.xml")//book/author)*

FOR *$a2* IN distinct-values *(doc("books.xml")//book/publisher)*

LET *$b := doc("books.xml")//book[author=$a1 and publisher = $a2]/price*

WHERE fn:exists*($b)*

RETURN {*$a1, $a2, avg($b)*}

Obviously, this XQuery is far from satisfactory. The first query needs one scan of the input document, the second and third query both need two times of scan, and the fourth query even requires three times.

To solve this problem, [57] introduced GROUP BY and NEST constructs. The following query provides the intended results, using this approach in conjunction with user-defined functions:

Example 5.2 *(XQuery with* GROUP BY *Implementation) Return the average price, (1) overall, (2) by author, (3) by publisher, and (4) by author-publisher combination.*

*/*An XQuery function to generate cube grouping dimensions*/*

DECLARE FUNCTION *local:cube ($dims* AS *item()*)* AS *item()* (*

```
IF empty ($dims) THEN <group/>
ELSE
  FOR $subgroup IN local:cube(fn:subsequence($dims, 2))
  RETURN ($subgroup,
    <group>{$dims[1], $subgroup/*}</group>))

/*The main XQuery which introduces new constructs GROUP BY and NEST*/
FOR $a IN doc("books.xml")/books/book
FOR $b IN $a/author
FOR $cell IN (local:cube($b, $a/publisher)
GROUP BY $cell
NEST $a/price into $price
RETURN
  <result>
    {$cell, <avg-price>avg($price)</avg-price>}
  </result>
```

The basic idea of this approach is as follows: first, a list of (author, publisher) pairs are generated. Then this list is passed to an XQuery function local:cube(), to generate grouping sets. For each input value pair *(author, publisher)*, the function will generate four items, $\emptyset$, *author*, *publisher*, and *(author, publisher)*, and store them *$cell* list. Then by the added constructs GROUP BY and NEST, prices for same items in $cell will be nested together into $price, and the average will be output in the return.

Although we only need one XQuery instead of four as in Example 5.1, the approach shown in Example 5.2 still has several problems, caused by XQuery's native function definition mechanism and the newly introduced constructs.

The native function definition problem arises from several aspects. First, there are several ways to write the XQuery UDF *local:cube()*, if the user chooses an inefficient definition, this will result in an inefficient query plan. This problem causes a difficult challenge for the query optimizer of the system, which must generate an optimal query plan out of an inefficient UDF definition. Furthermore, native XQuery processing engines available nowadays are very primitive compared to mature relational DBMS, the new grouping constructs will only exacerbate this primitivity. Second, suppose there are n *(author, publisher)* value pairs passed to the UDF, then the UDF will generate $4n$ items in the output, which will then be applied on the grouping query. This amounts to four scans of the original n pairs, which does not have any advantage over the previous queries in Example 5.1.

On the other hand, the new constructs GROUP BY and NEST introduce a lot of modifications to current XQuery evaluation models, which is difficult for commercial vendors, because of the upgrades required for the current implementations. Basically, a new query plan specific to such constructs needs to be designed to manage the element list in the GROUP BY, and the nested list for each group. Actually as discussed in [57], ORDER BY and WHERE clauses also need to be supported in the group clause, which introduces more complexity in query optimization. Efficiency of such query plans is also an open issue.

5.2 Grouping Sets Query Support

As in traditional OLAP, we refer to the components of XML analysis as analysis dimensions. We are mainly interested in two OLAP models. One is value-based model, which is like SQL OLAP queries, where analysis dimensions are decided by common values of relational columns, or values of attributes and/or text nodes

in XML's case. The other is unique to XML, which we call structure-based model, where analysis dimensions are decided by common structure, for example, nodes of same level or nodes of same path from the root.

5.2.1 Value-based OLAP Queries

The examples shown in Section 5.1 is a typical value-based OLAP query. Imagine that we have a relational data model, with 3-attribute schema *R (author, publisher, price)*, then the query Example 5.1 is exactly a cube query, which is easily expressed with the following SQL statement:

SELECT *author, publisher, avg(price)*
FROM *R*
GROUP BY *CUBE (author, publisher)*

With optimal index structure and robust query plan, such query will be very efficiently executed. The problem for XQuery, as in Example 5.2, however, is specific to the function definition mechanism provided by XQuery and does not affect the core of the language (i.e., the set of constructs used in defining the FLOWR expressions). Therefore, we keep the FLOWR constructs intact and investigate mechanisms to support more advanced functions in XQuery. Our solution consists in taking advantage of the fact that XQuery can accept functions defined in external languages and these can also be aggregate functions. In other systems, the functions to be imported into XQuery would be written in C/C++ or Java; but our system closely integrates XQuery and SQL, thus allows the importation of SQL:2003 analytical functions. This represents a significant improvement from the viewpoints of the user and the system. Namely, we extend XQuery by allowing it to declare and call SQL OLAP functions, thus support those complex OLAP queries in XQuery.

The examples in Example 5.1 with our approach, can be answered as in Example 5.3.

Example 5.3 *Return the average price (1) overall, (2) by author, (3) by publisher, and (4) by author-publisher combination.*

```
/*An XQuery function which uses external SQL/XML for CUBE query*/
DECLARE Function myCube
( ($author AS xs:string, $publisher AS xs:string, $price AS xs:float)* )
AS ELEMENT (group)*
{
      RETURN SQLXML (
        SELECT XMLElement (Name "group",
          XMLElement (Name "author", T.author),
          XMLElement (Name "publisher", T.publisher),
          XMLElement (Name "avg-price", avg(T.price)))
        FROM TABLE ($author, $publisher, $price)
          AS T (author, publisher, price)
        GROUP BY CUBE (T.author, T.publisher) )
}

/*The main XQuery which follows current standard*/
LET $r := {
      FOR $a IN doc("books.xml")/books/book
      FOR $b IN $a/author
      RETURN ($b, $a/publisher, $a/price) }
RETURN myCube($r)
```

Since each book might have multiple authors, we use a variable b to locate

every author, pair him/her with corresponding book's publisher and price, and generate a 3-field triplet. All such triplets are concatenated together in variable $r and passed to the external function myCube().

All the grouping aggregation is done in the aggregate function myCube(), with keyword SQLXML specifying it is defined in SQL/XML. Keyword TABLE is predefined to map the incoming XML elements into a flat relational table, so every triplet in $r will become one row in table T. The CUBE query itself is self-explanatory.

As a result, if there are *n (author, publisher)* value pairs, only *n* pairs will be passed to run the Cube query, which is more efficient than the *4n* case in Example 5.2.

Although XQuery does not have the notion of tuples, the correspondence between XML elements and the relational like tuple ($b, $a/publisher, $a/price) is not hard to maintain. Even if for the case that a book has multiple publishers or no publisher, a slight change of the main XQuery will cater to this different case. Alternatively, we can also predefine function myCube() in the system, taking variable number of parameters, and shield the code details from the users.

Similarly, all OLAP function constructs such as ROLLUP, CUBE, GROUPING SET, as well as HAVING, ORDER BY clause in SQL for filtering and output purposes, can be freely used here to satisfy the user's requirements.

From the user's point of view, he/she can directly use current standard to write XQuery which imports an external function, where he/she can also directly use SQL's OLAP grouping constructs to define any grouping dimensions. In this sense, the user does not need to learn any specific syntax to extend current XQuery or SQL for XML specific OLAP queries.

For implementation, we can use SQL's mature query optimization techniques

to optimize the OLAP performance. Specifically, the input XML data will be parsed to SAX event streams, which is essentially 3-field relational data stream. Since the OLAP analytics are still maintained in the SQL side, we can use efficient index structure to speed up the OLAP query. Thus, we avoid the need to develop XQuery-specific optimizations for OLAP queries.

In summary, we still keep the current XQuery FLWOR standard intact, without any new additions. Furthermore, our approach allows writing OLAP queries naturally, where the grouping dimensions are clearly expressed by SQL's grouping constructs. Techniques in relational data model are reused in our framework, which saves effort of developing new query plans for OLAP in native XQuery.

5.2.2 Structure-based OLAP Queries

XML's hierarchical structure allows for some unique OLAP queries, which are absent in the flat data model of relational tables. As an example, in the XML message for book sales as in Figure 5.1, an element "introduction" can appear in the fourth level or fifth level of the document. Consider a query such as, what is the book's average "introduction" length per "introduction" level? This question requires the use of structure-based analysis dimension. Obviously, it is hard to answer this question with current XQuery standard. The introduction of GROUP BY constructs, [70] allows expressing this query as follows:

Example 5.4 *(XQuery with* GROUP BY *Implementation) Return the average introduction length per introduction level.*

LET *$s, $s1, $s2 = Section*
FOR *$a* IN *(doc("books.xml")/books/book)*

GROUP BY *//{$s | $s1/$s2}/Introduction* INTO *$Intro*
NEST *($s1 == NULL ? 4 : 5)* INTO *$l, length($Intro)* INTO *$len*
RETURN {*$l, avg($len)*}

As discussed before, such XQueries require many modifications to the current XQuery evaluation models. Furthermore, we need to explicitly distinguish grouping based on node value, as in Section 5.2.1 and grouping based on structure, as in this example. More severe problems occur if "Introduction" elements can appear in any level of the XML documents, thus disallowing the enumeration of all possibilities as we did in the NEST clause in this example.

Our approach however, allows importing external SQL/XML statements from XQuery functions, thus this query can be easily answered by the following:

Example 5.5 *Return the average introduction length per introduction level.*

```
/*An XQuery function which uses external SQL/XML
DECLARE FUNCTION myLevelOLAP( ($level as xs:integer, $length as xs:integer)* )
AS ELEMENT(group)*
{
      RETURN SQLXML (
       SELECT XMLElement (Name "group",
        XMLElement (Name "level", T.level),
        XMLElement (Name "avg-length", avg(T.length)))
       FROM TABLE ($level, $length) AS T(level, length)
       GROUP BY T.level)
}

/*The main XQuery which follows current standard*/
```

```
LET $r := {
    FOR $a IN DOC("books.xml")/books/book//Introduction
    RETURN (fn:SAX-level($a), length($a)) }
RETURN myLevelOLAP ($r)
```

The XQuery still follows a simple format. Basically, a system library function fn:SAX-level() is predefined, to get the level of every "Introduction" element. Such system function can be efficiently implemented with SAX execution model, where a runtime stack is maintained to keep the depth of the current active elements. Then the level and length are passed to the external function, to calculate the average length of each level.

Such external function importing approach can also be conveniently used to express other structure-based OLAP queries, such as grouping dimensions based on path. For example, notice that in Figure 4.1, a book can belong to several categories, such as "software", "software/db", "software/db/concurrency", "distributed", etc), we can issue a similar query to return the average book price for each category. Due to space limitations, we omit the details here.

5.2.3 Structured Output Generation

In relational schema, the result of any SQL query is relational tuples, which is not the best solution from data publishing's point of view. Since we have supported OLAP queries on XML data model, it is essential for us to be able to generate meaningful hierarchical result as output of the OLAP queries. With SQL/XML, where constructs such as XMLElement, XMLAttributes, and XMLAgg are mainly used to publish relational query result as hierarchical output, it is easy to achieve such a data publishing goal.

We can use a complete example to illustrate our OLAP query and result publishing support efforts on XML data model. Suppose for the book purchasing web store, their sales statistics around the nation is encoded in XML, as Figure 5.2.

```
<sales_statistics>
  <region name = "PacificCoast">
    <state name = "CA">
      <city name = "Los Angeles">
        <sales>2400</sales>
      </city>
      <city name = "San Francisco">
        <sales>3000</sales>
      </city>
      ...
    </state>
    <state name = "WA">
      <city name = "Seattle">
        <sales>2500</sales>
      </city>
    </state>
    ...
  </region>
  <region name = "..." > ... </region>
  ...
</sales_statistics>
```

Figure 5.2: sales.xml: Business statistics for sales around the nation

Suppose we want to calculate the total sales, for every region, every state, and every city. This is a typical rollup query, with "ROLLUP" construct on relational model. With the approach we introduced in Section 5.2.1, this query can be answered by the XQuery statement in Example 5.6.

Example 5.6 *Return the total sales amount, by region, state, and city.*

```
/*An XQuery function which uses external SQL/XML for ROLLUP query*/
DECLARE FUNCTION myRollup ( ($region AS xs:string, $state AS xs:string, $city AS xs:string,
$sales AS xs:int)*) AS ELEMENT (group)*
{
      RETURN SQLXML (
       SELECT XMLElement (Name "group",
        XMLElement (Name "region", T.region),
        XMLElement (Name "state", T.state),
        XMLElement (Name "city", T.city),
        XMLElement (Name "total", sum(T.sales)))
       FROM TABLE ($region, $state, $city, $sales)
        AS T (region, state, city, sales)
       GROUP BY ROLLUP (T.region, T.state, T.city) )
}

/*The main XQuery which follows current standard*/
LET $r := {
      FOR $a IN Doc("sales.xml")/region
      FOR $b IN $a/state
      FOR $c IN $b/city
      RETURN
        ($a/@name, $b/@name, $c/@name, $c/sales) }
RETURN myRollup ($r)
```

As a result, disregarding the tag names, the structure of the output looks like a flattened relational data model, as follows, where '_' stands for NULL.

(PacificCoast, CA, Los Angeles, 2400)

(PacificCoast, CA, San Francisco, 3000)

(PacificCoast, CA, _ , 5400)

...

(PacificCoast, _ ,_ , 7900)

...

Figure 5.3: Output of Example 5.6 disregarding tag names

Suppose the user wants the output to be in a hierarchical format, which is more natural for a rollup query output, as in Figure 5.4.

```
<result>
  <region value = "PacificCoast">
    <sales_total>7900</sales_total>
    <state value = "CA">
      <sales_total>5400</sales_total>
      <city value = ``Los Angeles">
        <sales_total>2400</sales_total>
      </city>
      <city value = ``San Francisco">
        <sales_total>3000</sales_total>
      </city>
      ...
    </state>
    ...
  </region>
  ...
</result>
```

Figure 5.4: A sample structured output for ROLLUP query

To cater to this output requirement, we modify the external function of Example 5.6 as follows. In the first SQL statement, we define a ROLLUP view (r, s, c, sales_total) as in Figure 5.3, then in the second statement, we pass this 4-field

view together with their element names which will appear in the XML result, into a system built-in ROLLUP_output () function.

Example 5.7 *XQuery function to generate the output as Figure 5.4.*

```
DECLARE FUNCTION myRollup ( ($region AS xs:string, $state AS xs:string, $city AS xs:string, $sales AS xs:int)*) AS ELEMENT (result) {
    RETURN SQLXML (
     CREATE VIEW T_View (r, s, c, sales_total) AS
      SELECT T.region, T.state, T.city, sum(T.sales)
      FROM TABLE ($region, $state, $city, $sales)
        AS T (region, state, city, sales)
      GROUP BY ROLLUP (T.region, T.state, T.city) ;

     SELECT ROLLUP_output
        (r, 'region', s, 'state', c, 'city',
         sales_total, 'sales_total')
     FROM T_View )
}
```

Such Rollup_output() function is predefined in our system, which takes n (where n $= 4$ for example 5.7) pairs of parameters as input. The first $n-1$ (for example, n $=$ 3 in example 5.7) pairs are $n-1$ dimension values and names which will construct the grouping sets. The last pair (sales_total, 'sales_total') is the aggregation value for each group, and the element name it appears in the output.

Pseudo code to define the system function Rollup_output(), which uses SQL/XML's XMLElement, XMLAttributes, and XMLAgg is provided below. Such function is predefined in the system, thus the users are shielded from the implementation details.

Example 5.8 *Pseudo-code of Rollup structured output function.*

```
DECLARE FUNCTION ROLLUP_output (dim_list[ ], value, value_name)
{
1  TABLE temp (dim_list[ ], value, value_name);
2  INITIALIZE:
3  ITERATE: {
4     dim_value = dim_list[1] ;
5     dim_name = dim_list[2] ;
6     rest_dimensions = subsequence (dim_list, 3);
7     INSERT INTO temp VALUES (rest_dimensions, value, value_name)
8     WHERE at least one field in rest_dimensions is not null ;
9     INSERT INTO RETURN (
10          SELECT XMLElement (Name dim_name, XMLAttributes (dim_value AS "value"),
11               XMLElement (Name value_name, value),
12               XMLAgg (SELECT ROLLUP_output(rest_dimensions, value, value_name)
13                         FROM temp))
14                         WHERE every field in rest_dimensions equals to NULL);
15    DELETE FROM temp
16    WHERE every field in rest_dimensions equals to NULL;}
17    TERMINATE: {} }
```

From the pseudo-code of function *ROLLUP_output()*, we can see that it is actually a recursive aggregate function. Basically, such an aggregate function is specified by providing definition of an INITIALIZE, an ITERATE, and a TERMINATE computation, in a single procedure written in SQL. We assume that the data is ordered as in Figure 5.3, when it is passed to the aggregate function, otherwise we need to perform necessary reordering. The kernel of the aggregate function is in the ITERATE block, where in line 7-8 we store every input tuple

into a temporary table *temp*, until we see an input with NULL value in each field of *rest_dimensions*, which is used to construct the topmost elements in the return – line 9 to 14. Basically, in line 10, we take the first pair *(dim_value, dim_name)* from the dimension list, that is *('PacificCoast', 'region')* in this case, and use it to construct the top elements, such as *<region value = "PacificCoast">* in Figure 5.4. Then at line 11, the *value_name* and *value* are used to construct the first subelement of the outpost element, such as *<sales_total>7900</sales_total>* in Figure 5.4. Then in line 12 to 14, we recursively call *ROLLUP_output()* with the *rest_dimensions* in table *temp* to construct the subtrees. Finally in line 15 to 16, we delete everything in the *temp* table for later groups of the outmost elements.

Similar aggregate functions can be defined in the library, for other OLAP queries such as CUBE and GROUPING SET, to construct specific structured output for these queries.

5.3 Moving Windows Query Support on XML Streams

Streaming queries on XML data streams are becoming very popular in real-life applications, such as publish/subscribe and XML filtering systems, where the input and output should be pipelined based on a set of queries. XQuery is effective at expressing simple streaming applications, or applications on stored data. But it is not suitable for continuous applications, due to its blocking function definition mechanism, which needs to wait for the whole input element list before returning any result. In contrast, typical continuous queries require the invocation of a non-blocking function, where we only have a limited buffer, and incremental result needs to be outputted continuously, such as analytics on moving window, and pattern detection. The function importation mechanism discussed above is also applied to enable continuous queries over XML data. In this section, we

discuss how we extend XQuery to solve this problem, by allowing it to import non-blocking SQL:2003 functions or user-defined-aggregates (UDAs).

We still base our use cases on the hypothetical online web store for book purchases, as shown in Figure 5.1.

The first continuous query (CQ) application discussed here is moving-window calculation. Windows play a major role in relational DSMS and are supported by most data stream systems. SQL:2003 also supports some constructs to specify the moving windows.

Suppose we want to return the current average book price, every time when we get a new sale message posted. Without an efficient way to construct the unbounded moving window, a naive XQuery will be as follows:

Example 5.9 *Continuously return the average price (Naive Approach).*

```
LET $a := doc("books.xml")//book/price
FOR $s1 AT $i IN $a
RETURN
    {avg (FOR $s2 AT $j IN $a
      WHERE $j <= $i
      RETURN $s2)}
```

The meaning of this XQuery is not clear, since the window is manually constructed inside the avg aggregate. Furthermore, it is not a real streaming query, because it constructs a whole bid price list $a, reads it into memory, and uses two cursors ($i, $j) to scan $a to control the window boundary. In addition, the average is recomputed over the window for each new price element, as opposed to performing delta computation.

However, such moving window calculation is very easy to write in relational

data model, using SQL:2003's constructs such as PRECEDING. Implementation-wise, efficient query plans are available for such continuous queries on streaming data, where only O(1) buffer space is needed to keep temporary result and return the moving average. Thus, we aim to extend XQuery to call external functions, which are actually SQL:2003 statements, to achieve greater user-friendliness, a leap in expressive power, and efficient query execution, as follows:

Example 5.10 *Continuously return the average price (Our Approach).*

```
/*An XQuery function which imports external SQL/XML on an unbounded
moving window*/
DECLARE FUNCTION myCountWindow($price AS xs:real)
AS ELEMENT (avg-price)
{
   RETURN SQLXML (
      SELECT XMLElement (Name "avg-price", avg(T.price)
       OVER (ROWS UNBOUNDED PRECEDING))
     FROM TABLE ($price) AS T (price) )
}

/*The main XQuery which follows the current standard*/
FOR $a IN doc("books.xml")/books/book
LET $b := $a/price
RETURN myCountWindow($b)
```

Basically, we use FOR clause in the main XQuery, so instead of returning one overall average price as those OLAP example in Section 3, this query will return the current average price for every incoming book sale message. The external function myCountWindow() contains a straightforward SQL:2003 query.

Example 5.10 is a count-based physical window operation with XQuery. Example 5.11 is another important continuous query application, namely, time-based logical window operation.

Example 5.11 *Continuously return the average price of last 5 minutes, grouping by every publisher.*

```
/*An XQuery function which uses external SQL/XML on a 5-minute moving window*/
DECLARE FUNCTION myTimeWindow($publisher as xs:string, $price as xs:float, $posttime as
xs:timestamp)
AS ELEMENT (avg-for-last-5-min)*
{
      RETURN SQLXML (
       SELECT XMLElement (Name "avg-for-last-5-min",
        XMLElement (Name "post_time", T.posttime),
        XMLAgg (
         XMLElement (Name "publisher", T.publisher),
         XMLElement (Name "avg-price", avg(T.price))
         OVER (PARTITION BY T.publisher RANGE 5 MINUTES PRECEDING)))
       FROM TABLE($publisher, $price, $posttime)
        AS T(publisher, price, posttime) )
}
/*The main XQuery which follows the current standard*/
FOR $a IN doc("books.xml")/books/book
LET $b := $a/publisher
LET $c := $a/price
LET $d := $a/posttime
RETURN myTimeWindow($b, $c, $d)
```

This XQuery is very similar to that of Example 5.10. Basically we construct a relational stream ($b, $c, $d) and pass it to the external function. In the external function myavg(), we can use any SQL:2003 constructs, such as the PARTITION BY, PRECEDING, to specify the grouping functionality for every book and moving window for the last five minutes.

In addition to the built-in aggregate functions such as avg() and count(), we also extend XQuery to call more complex user-defined aggregates (UDA) on moving windows. As a result, we support advanced sequence and time-series queries that represent another important application area for continuous queries, such as pattern detection.

5.4 Performance Study

The motivation of our approach is not only minimum language extensions, easy expression, but also better query performance. By extending XQuery functions to directly use OLAP query optimization techniques in relational database system, such as Stream Mill in our implementation, we can show tremendous performance advantage over equivalent XQueries issued on native XML database, or relational database products integrated with XML extensions following current XQuery standard.

Our experiments are based on large XML documents keeping history information of a transaction database for employees, which records the change of salary, title, and department for each employee. Basically, an employee element can have multiple salary subelements, ordered by the valid time of the values. The same case is applicable for title and dept subelements. Sample schema is as Figure 5.5.

Interesting OLAP queries and moving window queries can be issued on such

```
<employees>
  <employee>
    <salary tstart = "02-20-98" tend = "02-19-99">40000</salary>
    <salary tstart = "02-20-99" tend = "02-04-00">42100</salary>
    <salary tstart = "02-05-00" tend = "02-04-01">42525</salary>
    ...
    <title tstart = "02-20-98" tend = "07-01-00">Software Engineer</title>
    <title tstart = "07-02-00" tend = "07-01-05">Sr. Software Engineer</title>
    <dept>...</dept>
    <dept>...</dept>
  </employee>
  ...
</employees>
```

Figure 5.5: A sample XML document for employee history information

XML document, such as the total employee number in each dept, or the average salary in the past 2 years for every employee.

We generate a data set to model the history of employees over 17 years, 20 departments, 4 possible titles. The total size of the published XML documents is around 230MB.

We base our query performance comparison on three different systems. The first system is our XQuery extension on top of Stream Mill system [28]. The second system uses IBM DB2 with its latest XML extension. The third system is native XML database system Quip [6].

We run six sample queries, with different grouping dimensions, resulting in different number of aggregation values, as follows.

- Q1: Current average salary for every dept (20 groups, one aggregation value per group).
- Q2: Current average salary, total number of employees, for every dept (20 groups, two aggregation value per group).

- Q3: Current average salary overall, by dept, by (dept, title) (ROLLUP query, 101 groups, one aggregation value per group).
- Q4: Current average salary, overall, by dept, by title, by (dept, title) (CUBE query, 101 groups, one aggregation value per group).
- Q5: Current average salary, total number of employees, by dept, by title, by (dept, title) (CUBE query, 101 groups, two aggregation value per group).
- Q6: Moving average salary every year, for every dept (moving window query, 340 groups, one aggregation value per group).

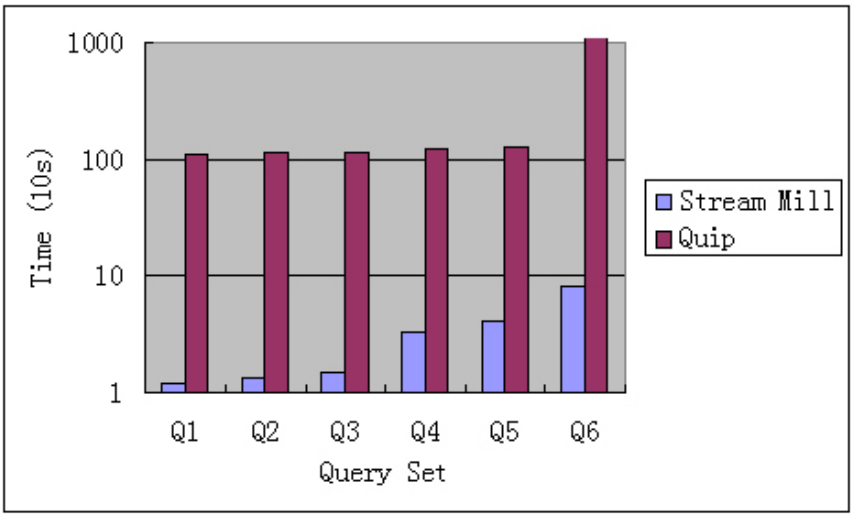

Figure 5.6: Performance comparison of STREAM MILL and Quip

Figure 5.6 shows the substantial performance advantage of our approach over Quip. Several reasons explain such difference. One is the different query efficiency between two set of queries. In the native XML system, because it only supports current XQuery standard, we need to write several XQuery statements for a certain query, which introduces a lot of scans over the original large document. The other reason is the mature query optimization we can use in a relational engine.

Even for a query which scans the same number of times in both systems, the relational indexing techniques in Stream Mill beats its counterpart in XML. This is not surprising because this is a general difference between relational database and native XML database.

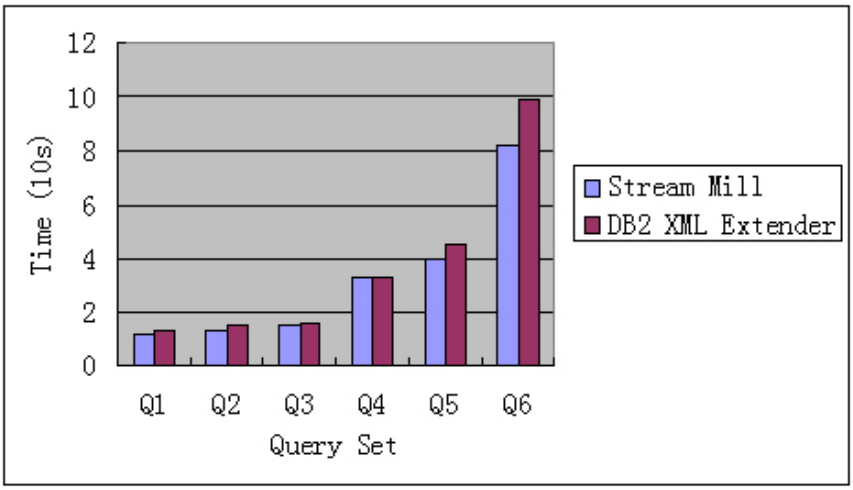

Figure 5.7: Performance comparison of STREAM MILL and DB2 XML extension

Figure 5.7 shows the difference between our approach and corresponding XQueries on DB2's XML extension. Although this time our performance advantage is not so obvious as compared to Quip, we still beat DB2 XML extension in every query set. The extra overhead occurred in DB2 is because it decomposes the XML document into several child tables, and its query translation will introduce some extra joins among several tables. As a contrast, our XQuery compiler compiles the XQueries to several SQL-based statements which can be executed in parallel on the incoming SAX streams.

We further test the scalability of our system on query Q6, on 1/4, 1/2, 3/4, and whole data set. Figure 5.8 shows that our algorithm is linearly scalable to the increase in data size, which conforms to its single scan feature.

In summary, our approach of allowing XQuery functions to import SQL OLAP

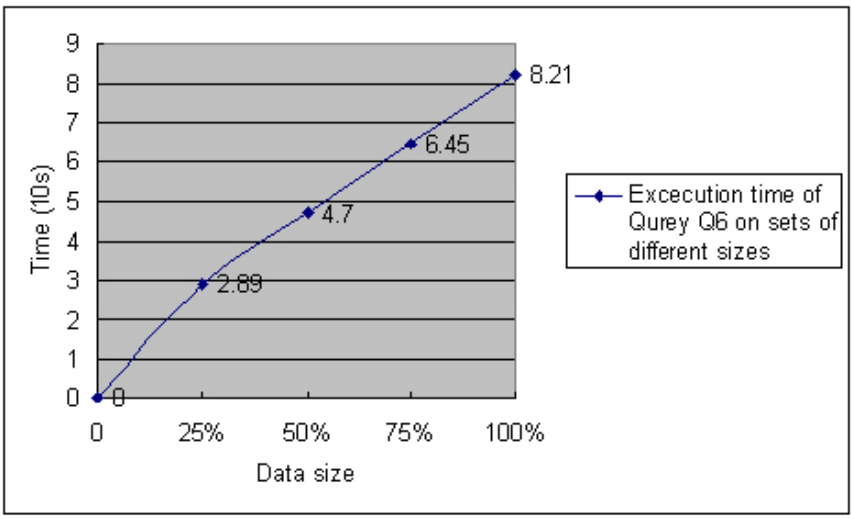

Figure 5.8: Execution time of Query Q6 on data sets of different sizes

analytics is easily implemented on top of Stream Mill, and gains obvious performance advantage over native XML database Quip, as well as DB2 XML extension.

CHAPTER 6

Conclusion

The integration of relational databases and XML has become more and more important due to the rapid development of XML-based information exchange and manipulation applications. New XML data are to be stored, queried, and manipulated in the database along with existing relational data; archival and history relational data also need to be published as XML. Furthermore, new applications are emerging for streaming data, where input and output are pipelined as relational data streams or XML data streams. Support for these two kinds of streams should be integrated into a data stream management system, which incorporate efficient streaming processing techniques that are in great need for online applications. Online Analytical Processing (OLAP) has proved to be a valuable tool for analyzing trends in business information, and similar tools for analyzing XML documents are urgently needed as XML is becoming the language of choice for data representation across a wide range of application domains.

In this dissertation, we have made contributions to three critical research areas for XML, including: (i) temporal information management using XML, (ii) streaming XML processing, and (iii) OLAP extensions for XML query languages.

In temporal information management applications, XML provides a good representation for temporal-grouped data information, upon which an H-view can be defined for the history of a relational database. On the other hand, different relational views and storage models cater to different applications. We explored

three logical view models as follows: the XML-based view, the nested-relational view, and the event-history-based relational view. We also studied typical temporal queries on each of these models as well as their efficient implementation. In particular, we showed that temporal coalescing queries can have simple and efficient solutions with either SQL:2003 OLAP functions or user-defined-aggregates (UDAs).

The second focus of this dissertation is in the area of integrating streaming XML data with relational data streams. We achieved this objective by flattening XML streams into relational streams, with the help of simple UDAs. We also leveraged the expressive power of UDAs to support all kinds of advanced continuous queries on the integrated streams. By using a UDA to simulate a FSA, we can implement FSA-based multi-query processing, with performance that is comparable with specialized systems such as YFilter. We integrated all of these functionalities in a powerful general-purpose data stream system prototype, Stream Mill.

The third contribution in this dissertation is OLAP support on XML data. We showed how XQuery can be extended to support complex grouping sets, as well as moving windows, which are missing in current XQuery standards. After reviewing the shortcomings of current proposed approaches which introduce new constructs into XQuery standards, we provided the approach whereby XQuery can import SQL/XML functions. As a result of this extension, all complex grouping set queries and moving windows can be easily specified using SQL:2003, and good query performance can be expected because we can build on mature relational technology to support OLAP queries efficiently.

References

[1] Database Languages SQL, ISO/IEC 9075-*:2003.

[2] DB2 XML Extender. World Wide Web, `http://www-3.ibm.com/software/data/db2/extenders/xmlext/index.html`.

[3] Informix universal server. http://www.ibm.com/informix.

[4] Oracle flashback technology, http://otn.oracle.com.

[5] Oracle9i XML Database Developer's Guide - Oracle XML DB Release 2 (9.2). World Wide Web, `http://otn.oracle.com/tech/xml/xmldb/content.html`.

[6] Quip. World Wide Web, `developer.softwareag.com/tamino/quip`.

[7] SQL 2003 Standard Support in Oracle Database 10g. World Wide Web, `otn.oracle.com/products/database/application\_development/pdf/SQL\_2003\_TWP.pdf`.

[8] SQL/XML. World Wide Web, `http://www.sqlx.org`.

[9] The Extensible Stylesheet Language (XSL). http://www.w3.org/Style/XSL/.

[10] XML Path Language (XPath) Version 1.0. World Wide Web, `http://www.w3.org/TR/xpath/`.

[11] XML Protocols. World Wide Web, `http://www.w3.org/2000/xp/`.

[12] XML Schema. World Wide Web, `http://www.w3.org/XML/Schema/`.

[13] XQuery 1.0: An XML Query Language. World Wide Web, `http://www.w3.org/TR/xquery/`.

[14] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT Framework for Logical XQuery Optimization. In *VLDB*, 2004.

[15] A. Eisenberg, J. Melton, K. Kulkarni, et al. SQL:2003 Has Been Published. In *SIGMOD Rec.*, volume 33, pages 119–126, 2004.

[16] M. Altinel and M. Franklin. Efficient filtering of xml documents for selective dissemination of information. In *VLDB*, 2000.

[17] A. Arasu, S. Babu, and J. Widom. An abstract semantics and concrete language for continuous queries over streams and relations. Technical report, Stanford University, 2002.

[18] D. Barbara. The characterization of continuous queries. *Intl. Journal of Cooperative Information Systems*, 8(4):295–323, 1999.

[19] C. Barton, P. Charles, and D. Goyal, et al. Streaming xpath processing with forward and backward axes. In *ICDE*, 2003.

[20] K. Bayer, R. J. Cochrane, V. Josifovski, and et. al. System RX: One Part Relational One Part XML. In *SIGMOD*, 2005.

[21] E. Bertino, E. Ferrai, and G. Guerrini. A formal temporal object-oriented data model. In *EDBT*, 1996.

[22] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo. Coalescing in Temporal Databases. In *VLDB*, 1996.

[23] R. Bourret. XML and Databases. World Wide Web, `http://saxproject.org`.

[24] N. Bruno and L. Gravano, et al. Navigation- vs. index-based xml multi-query processing. In *ICDE*, March 2003.

[25] C. A. Hurtado, A. O. Mendelzon. Reasoning about Summarizability in Heterogeneous Multidimensional Schemas. In *ICDT*, 2001.

[26] C. Koch, S. Scherzinger, N. Schweikardt, and et al. Schema-based Schedulng of Event Processors and Buffer Minimization for Queries on Structured Data Streams. In *VLDB*, 2004.

[27] C. Luo, H. Thakkar, H. Wang, and C. Zaniolo. A Native Extension of SQL for Mining Data Streams. In *SIGMOD Demo*, pages 873–875, 2005.

[28] C. Zaniolo, R. Luo, H. Wang, et al. Stream Mill: Bringing Power and Generality to Data Stream Management Systems. World Wide Web, `http://wis.cs.ucla.edu/stream-mill/index.html`.

[29] E. Camossi, E. Bertino, G. Guerrini, and M. Mesiti. Automatic evolution of multigranular temporal objects. In *TIME*, 2002.

[30] M. Carey, J. Kiernan, J. Shanmugasundaram, and et al. XPERANTO: A Middleware for Publishing Object-Relational Data as XML Documents. In *VLDB*, 2000.

[31] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, Hong Kong, China, 2002.

[32] C. Chan and P. Felber, et al. Efficient filtering of xml documents with xpath expressions. In *ICDE*, 2002.

[33] S. Chandrasekaran and M. Franklin. Streaming queries over streaming data. In *VLDB*, 2002.

[34] C. Chatfield. *The Analysis of Time Series: An Introduction.* Chapman & HallCRC, 1996.

[35] S. Chaudhuri and U. Dayal. "an overview of data warehousing and olap technology". In *Data Mining and Knowledge Discovery*, 1997.

[36] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchical structured information. In *SIGMOD*, 1996.

[37] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, pages 379–390, May 2000.

[38] J. Clifford, A. Croker, F. Grandi, and A. Tuzhilin. On temporal grouping. In *Recent Advances in Temporal Databases*, pages 194–213. Springer Verlag, 1995.

[39] J. Clifford, C.E. Dyreson, T. Isakowitz, C.S. Jensen, and R.T. Snodgrass. On the Semantics of "Now" in Databases. *TODS*, 22(2):171–214, 1997.

[40] D. Pedersen, K. Riis, and T.B. Pedersen. Query Optimization for OLAP-XML Federations. In *DOLAP*, 2002.

[41] D. DeHaan, D. Toman, M. P. Consens, and M. T. Ozsu. A Comprehensive XQuery to SQL Translation Using Dynamic Interval Encoding. In *SIGMOD*, 2003.

[42] Yanlei Diao and Michael J. Franklin. Query processing for high-volume xml message brokering. In *VLDB*, 2003.

[43] A. Marian et. al. Change-centric management of versions in an xml warehouse. In *VLDB*, 2001.

[44] F. Peng and S. S. Chawathe. XPath Queries on Streaming Data. In *SIGMOD*, 2003.

[45] F. Wang and C. Zaniolo. Publishing and Querying the Histories of Archived Relational Databases in XML. In *WISE*, 2003.

[46] M. F. Fernandez, A. Morishima, D. Suciu, and W. C. Tan. Publishing Relational Data in XML: the SilkRoute Approach. *IEEE Data Engineering Bulletin*, 24(2):12–19, 2001.

[47] Daniela Florescu, Chris Hillery, and Donald Kossmann, et al. *The bea/xqrl streaming xquery processor*. 2003.

[48] et al G. Gray, S. Chaudhuri. "data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals". In *ICDE*, 1996.

[49] Lukasz Golab and M. Tamer zsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, 2003.

[50] F. Grandi. An Annotated Bibliography on Temporal and Evolution Aspects in the World Wide Web. In *TimeCenter Technique Report*, 2003.

[51] A. K. Gupta and D. Suciu. Streaming processing of xpath queries with predicates. In *SIGMOD*, June 2003.

[52] S. Holzner. *Inside XML*. New Riders Publishing, 2000.

[53] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Lanaguages and Computation*. 1979.

[54] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. An xml query engine for network-bound data. In *VLDB Journal*, 2002.

[55] H. Jagadish, I. Mumick, and A. Silberschatz. View maintenance issues for the chronicle data model. In *PODS*, pages 113–124, 1995.

[56] M. R. Jensen, T. H. Moller, and T. B. Pedersen. Specifying OLAP Cubes on XML Data. In *Internaltional Conference on Scientific and Statistical Database Management*, 2001.

[57] K. S. Beyer, D. Chamberlin, L. Colby, et al. Extending XQuery for Analytics. In *SIGMOD*, 2005.

[58] K. S. Beyer, R.J. Cochrane, L.S. Colby, et al. XQuery for Analytics: Challenges and Requirements. In *XIME-P*, 2004.

[59] R.H. Katz and E. Chang. Managing Change in Computer-Aided Design Databases. In *VLDB*, 1987.

[60] S. Kepser. A proof of the turing-completeness of xslt and xquery. In *Technical report SFB 441, Eberhard Karls Universitat Tubingen*, 2002.

[61] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. Query languages and data models for sequences and streams. VLDB 2004.

[62] T.Y. Leung and H. Pirahesh. Querying Historical Data in IBM DB2 C/S DBMS Using Recursive SQL. In *Recent Advances in Temporal Databases*, 1995.

[63] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *TKDE*, 11(4):583–590, August 1999.

[64] M. R. Jensen, T. H. Moller, and T. B. Pedersen. Specifying OLAP Cubes on XML Data. In *Proceedings of the 13th International Conference on Scientific and Statistical Database Management*, 2001.

[65] Sam Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, pages 49–61, 2002.

[66] J. Naughton, et al. The Niagara Internet Query System. In *IEEE Data Engineering Bulletin*, 2001.

[67] G. Özsoyoğlu and R. Snodgrass. Temporal and real-time databases: A survey. *TKDE*, 7(4), August 1995.

[68] P. Vassiliadis, T. Sellis. A Survey of Logical Models for OLAP Databases. In *SIGMOD Record 28(4)*, 1999.

[69] N. Pendse. The OLAP Report. World Wide Web, `www.olapreport.com`.

[70] R. R. Bordawekar, C. A. Lang. Analytical Processing of XML Documents: Opportunities and Challenges. In *SIGMOD Record, Vol 34. No. 2*, 2005.

[71] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Expressing and Optimizing Sequence Queries in Database Systems. *TODS*, 29(2):282–318, 2004.

[72] Reza Sadri, Carlo Zaniolo, and Amir Zarkesh, et al. Optimization of sequence queries in database systems. In *PODS*, 2001.

[73] Reza Sadri, Carlo Zaniolo, and Amir Zarkesh, et al. Expressing and optimizing sequence queries in database systems. In *TODS*, 2004.

[74] H. Schöning. Tamino - a DBMS Designed for XML. In *ICDE*, 2001.

[75] P. Seshadri, M. Livny, and R. Ramakrishnan. Seq: A model for sequence databases. In *ICDE*, pages 232–239, 1995.

[76] J. Shanmugasundaram and et al. Efficiently Publishing Relational Data as XML Documents. In *VLDB*, 2000.

[77] R. Snodgrass. Temporal object-oriented databases: a critical comparision. *Addions-Wesley/ACM Press*, 1985.

[78] R. T. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[79] R. T. Snodgrass. Developing time-oriented database applications in sql. *Morgan Kaufmann*, 1999.

[80] Richard T. Snodgrass, Michael H. Böhlen, Christian S. Jensen, and Andreas Steiner. Transitioning Temporal Support in TSQL2 to SQL3. *Lecture Notes in Computer Science*, 1399:150–194, 1998.

[81] M. Sullivan. Tribeca: A stream database manager for network traffic analysis. In *VLDB*, 1996.

[82] T. Fiebig and G. Moekotte. Algebraic XML Construction in Natix. In *WISE*, 2001.

[83] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *SIGMOD*, pages 321–330, 6 1992.

[84] D. Toman. Point-based Temporal Extensions of SQL. In *DOOD*, pages 103–121, 1997.

[85] F. Wang and C. Zaniolo. Representing and querying the evolution of databases and their schemas in xml. In *Intl. Workshop on Web Engineering, SEKE*, 2003.

[86] F. Wang and C. Zaniolo. Temporal Queries in XML Document Archives and Web Warehouses. In *TIME-ICTL*, 2003.

[87] F. Wang and C. Zaniolo. XBiT: An XML-based Bitemporal Data Model. In *ER*, 2004.

[88] F. Wang and C. Zaniolo. An XML-Based Approach to Publishing and Querying the History of Databases. *To Appear in World Wide Web: Internet and Web Information Systems*, 2005.

[89] F. Wang, X. Zhou, and C. Zaniolo. Using XML to Build Efficient Transaction-Time Temporal Database Systems on Relational Databases. Submitted to VLDB 2004.

[90] F. Wang, X. Zhou, and C. Zaniolo. Using XML to Build Efficient Transaction-Time Temporal Database Systems on Relational Databases. Technical Report 81, TimeCenter, www.cs.auc.dk/TimeCenter, Mar. 2005.

[91] H. Wang and C. Zaniolo. Using SQL to Build New Aggregates and Extenders for Object-Relational Systems. In *VLDB*, 2000.

[92] Haixun Wang and Carlo Zaniolo. Atlas: a native extension of sql for data minining. In *Third SIAM Int. Conference on Data MIning*, 2003.

[93] X. Zhou, H. Thakkar, Carlo Zaniolo. Extending XQuery for OLAP Functions and Continuous Queries. In *Submitted to VLDB*, 2006.

[94] X. Zhou, H. Thakkar, Carlo Zaniolo. Unifying the Processing of XML Streams and Relational Data Streams. In *ICDE*, April 2006.

[95] Y. Diao, M. Altinel, M. Franklin, et al. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. In *TODS*, 2003.

[96] C. Zaniolo, S. Ceri, C.Faloutsos, R.T. Snodgrass, V.S. Subrahmanian, and R. Zicari. *Advanced Database Systems*. Morgan Kaufmann Publishers, 1997.