2017

Online Integration of Semistructured Data

Handoko
University of Wollongong

Online Integration of Semistructured Data

A thesis submitted in fulfillment of the
requirements for the award of the degree

Doctor of Philosophy

from

UNIVERSITY OF WOLLONGONG

by

Handoko

School of Computer Science and Information Technology
January 2017

Dedicated to
My Family and my parents

Declaration

This is to certify that the work reported in this thesis was done by the author, unless specified otherwise, and that no part of it has been submitted in a thesis to any other university or similar institution.

Handoko
January 5, 2017

Abstract

Data integration systems play an important role in the development of distributed multi-database systems. Data integration collects data from heterogeneous and distributed sources, and provides a global view of data to the users. Systems need to process user's applications in the shortest possible time. The *virtualization approach* to data integration systems ensures that the answers to user requests are the most up-to-date ones. In contrast, the *materialization approach* reduces data transmission time at the expense of data consistency between the central and remote sites. The *virtualization approach* to data integration systems can be applied in either batch or online mode. Batch processing requires all data to be available at a central site before processing is started. Delays in transmission of data over a network contribute to a longer processing time. On the other hand, in an online processing mode data integration is performed *piece-by-piece* as soon as a unit of data is available at the central site. An online processing mode presents the partial results to the users earlier. Due to the heterogeneity of data models at the remote sites, a semistructured global view of data is required. The performance of data integration systems depends on an appropriate data model and the appropriate data integration algorithms used.

This thesis presents a new algorithm for immediate processing of data collected from remote and autonomous database systems. The algorithm utilizes the idle processing states while the central site waits for completion of data transmission to produce instant partial results. A decomposition strategy included in the algorithm balances of the computations between the central and remote sites to force maximum resource utilization at both sites. The thesis chooses the XML data model for the representation of semistructured data, and presents a new formalization of the XML data model together with a set of algebraic operations. The XML data model is used to provide a virtual global view of semistructured data. The algebraic operators are consistent with operations of relational algebra, such that any existing syntax based query optimization technique developed for the relational model of data can be directly applied. The thesis shows how to optimize

online processing by generating one online integration plan for several data increments. Further, the thesis shows how each independent increment expression can be processed in a parallel mode on a multi core processor system. The dynamic scheduling system proposed in the thesis is able to defer or terminate a plan such that materialization updates and unnecessary computations are minimized. The thesis shows that processing data chunks of fragmented XML documents allows for data integration in a shorter period of time.

Finally, the thesis provides a clear formalization of the semistructured data model, a set of algorithms with high-level descriptions, and running examples. These formal backgrounds show that the proposed algorithms are implementable.

Acknowledgements

First and foremost, all praise and thanks to Jesus Christ for the opportunity to pursue my PhD studies and learn many things during my studies.

I would like to express my sincere appreciation to my supervisor Dr. Janusz Getta, for his kindness, patient guidance, encouragement and the advice he has provided throughout my time as his student. I would like to thank him for his financial supports for attending some conferences, and all recommendations for my scholarship to finish my study.

I must thank all the members of the School of Computing and Information Technology, including the students; special thanks to all my friends in lab 3.234, for making such a friendly enlightening discussion.

To my many friends, you should know that your support and encouragement was worth more than I can express on paper. Special thanks to Gateway City Church Wollongong, for being my second home.

I must thank the Indonesian Government, and particularly the Directorate General of Higher Education (Dikti), Indonesian Ministry of National Education, for providing a scholarship for my PhD studies.

Last but not the least, I would like to thank my family for their unconditional support, both financially and emotionally throughout my study. In particular, the understanding shown by my beloved wife Mesky, my sons Jehoshua and Aaron, is greatly appreciated.

Publications

1. Handoko and Janusz R. Getta. An XML algebra for online processing of XML documents. In *The 15th International Conference on Information Integration and Web-based Applications & Services*, IIWAS '13, Vienna - December 2-4, 2013.

2. Handoko and Janusz R. Getta. Query Decomposition Strategy for Integration of Semistructured Data. In *The 16th International Conference on Information Integration and Web-based Applications & Services*, IIWAS '14, Hanoi - December, 2014.

3. Getta J.R., Handoko. *On Transformation of Query Scheduling Strategies in Distributed and Heterogeneous Database Systems*. In: Nguyen N., Trawiski B., Kosala R. (eds) Intelligent Information and Database Systems. ACIIDS 2015. Lecture Notes in Computer Science, vol 9011. Springer, Cham, 2015.

4. Handoko and J. R. Getta, Concurrent processing of increments in online integration of semi-structured data, *Information and Communication Technology (ICoICT), 2015 3rd International Conference on*, Nusa Dua, 2015, pp. 289-294. doi: 10.1109/ICoICT.2015.7231438

5. Handoko and Janusz R. Getta. Dynamic query scheduling for online integration of semistructured data. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual, volume 3, pages 375-380*, July 2015.

6. Handoko and Janusz R. Getta. Online Integration of Fragmented XML Documents. Accepted for publication in *9th Asian Conference on Intelligent Information and Database Systems (ACIIDS)*, April 2017.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

In modern data processing, there are many reasons why database systems are distributed over a network. First, data grows so fast that the volume of data is too large, and it is too expensive to provide computing hardware to manage the database. The second reason is to provide the user with better access by placing the database at the nearest location. Distributed database systems may start from a centralized database system which is then fragmented into several database systems for whatever reason. They are usually developed using homogeneous database systems, and with a well defined database schema. On the other hand, distributed database systems may start from independent database systems which are connected over a network in order to get a bigger picture of data. They are usually heterogeneous, since the external database sites connected over the network can be developed independently accordingly to their own characteristics. Distributed database systems are scalable, such that a new database node can be added to the network whenever it is needed.

Distributed database systems allow raw data to be scattered over the network, and sometimes with an undefined global schema. Despite the fact that being distributed creates some benefits, processing data over a network faces some obstacles. Data processing requires transmission costs which are unpredictable, the number of external sites are growing, and various data structures are used. As a consequence, an additional effort is required if a bigger picture of data is desirable.

A data integration system gathers data from different sources in order to provide answers to user queries. Data integration systems are classified into two different approaches, the *materialization approach* and the *virtualization approach*. A *materialization approach* data integration system collects data from multiple data sources and makes a copy of the data at the central site. Then, user queries are computed against materialized data at the central site. To maintain consistency

of the data sources, a materialized data must be refreshed from time to time either via an immediate or a delayed update. In an immediate update, any change to the data source is instantly applied to its corresponding materialized data. The data source plays an important role in order to keep materialized data up-to-date. On the other hand, an update to the materialized data is performed at a scheduled time or when it is desired. A delayed update may cause inconsistency between materialized data and the data sources.

In a *virtualization approach* data integration system, the central site provides a virtual global view of data over network. User queries to the central site are decomposed and sent to the external sites for computation. Then, the external sites transmit the results to the central site for further processing. In this approach, the up-to-date data in distributed database systems can be obtained. The biggest challenge of the *virtualization approach* to data integration is to provide data to user within a reasonable processing time.

Processing of a user request may be performed either in a batch or an online processing mode. In a batch processing mode, computation is performed on a collection of input data. Batch processing requires that all data be available at the central site before computation can be started. Since data integration systems involve large data, batch processing of user queries is frustrating and requires a longer time to get results.

Meanwhile, online processing allows data integration to be performed as soon as a unit of data is available at the central site. Online integration is a process of continuous consolidation of data transmitted over a network with the data already available at the central site of a distributed multi-database system. Online integration applies online processing where a unit of increment data is instantly processed without having the entire set of data available. Then, the result of the incremental data processing is combined with the current state to get a new state of processing. Online data integration takes advantage of online processing by utilizing of waiting time for data transmission, starts processing earlier, and therefore reduce processing time.

Another challenge of a data integration system is unification of heterogeneous data models. The global view of external database sites has to be good enough to give a uniform model of heterogeneous data in a distributed multi-database system. In the last decade, semistructured data has been used as a model for information exchange. The semistructured data model represents complex structures, is expandable, and is human-machine readable. XML has been widely used for representing semistructured data.

This chapter presents an introduction to the topic. Section 1.1 describes the motivation of the thesis. The research problem is stated in 1.2, then the thesis is outlined in Section 1.3.

1.2 The Problem Statement

In this thesis, we assume the *virtualization approach* to a data integration system where the external database sites are highly autonomous. The *virtualization approach* to data integration requires a global view of heterogeneous data which is inherently based on the concept of the relational model. Data integration based on the relational model is not suitable for semistructured data. Apart from that, using a batch processing mode in a data integration system requires a longer time to get the first results of the computation.

It is proposed that *online data integration* powered with a suitable data model and algebra can provide improved performance of a data integration system. The central question to be answered is how to develop an efficient data integration system with a global view of semistructured data.

To answer these problems, we decompose them into smaller sub problems as follows:

1. The system must provide a global view of semistructured data for online data integration. It can be solved by providing answers to the following problems:
   (a) How to design a formalization of the XML model for online processing.
   (b) How can a set of XML algebraic operators and rules for semistructured data be designed such that it is consistent with the relational algebra.
2. The system must provide *online data integration* to process user queries. Answers to the following problems lead to a solution:
   (a) How to provide a good query decomposition strategy to balance processing between the central site and the external sites.
   (b) How to design an online algorithm which allows processing of incremental data.
   (c) How to generate suitable execution plans depending on the circumstances of data increments.
   (d) How can a scheduling system provide a good strategy to execute all plans generated.

(e) How to efficiently process large size documents in the data integration system.

The purpose of this thesis is to address the problems stated above in the following ways:

1. In order to provide a global view of semistructured data for online processing, an approach to the formalization of an XML data model is presented. This approach is based on an extension of Regular Tree Grammar which allows one to build a structure where an element may be used in several levels of a document. The Extended Tree Grammar allows easy manipulation of the XML document's structure, such as document extraction, concatenation, and element removal. The features of Extended Tree Grammar play an important role when operations in a data integration expression are performed. Moreover, in processing large size XML documents, the Extended Tree Grammar supports the fragmentation operation of XML documents, as well as the defragmentation operation.

2. A new approach to the definition of an XML algebra for online processing is presented. XML algebra operators which are consistent with relational algebra are introduced. Operations on Extended Tree Grammar which allow modification of the XML document structure are presented. The operators of XML algebra process data containers. A data container is a set of XML documents, and is equivalent to a table in the relational model. The system includes the following basic operators: *projection, selection, join, antijoin* and *union*. The XML algebraic operators and their properties have a special emphasis which allows incremental processing of a data integration expression.

3. A query decomposition strategy is presened which considers the characteristics of all resources in a multi-network database system, including the remote sites, the central site, and the network. The approach balances query processing between the central and remote sites to improve the performance of processing. A *global query expression* is broken into sub-expressions and finds a set of sub-expressions whose overall processing requires the lowest cost. A cost function is introduced, as well as a set of algorithms, in order to achieve optimal query decomposition.

4. The problem of efficient data integration is tackled by exploiting algebraic properties of our operators. This permits the study of equivalence of data

integration expressions, and the production of a set of transformation rules which allows the processing of data integration in an online mode, where a unit of data increment is computed without waiting for the entire data to be available at the central site. The transformation rules produce an increment expression which combines the result of processing a unit of data increment with the previous result of data integration to obtain a new state of computation. The increment expression allows data integration to start its computation as soon as a unit of data arrives at the central site.

5. Transformation of the increment expressions into an online integration plan is shown by mapping every simple algebraic expression into corresponding step. The transformation starts from the inner-most XML algebraic operation. The online integration plan includes operations to update the materialized results of computation, and is generated for every data container involved in a data integration expression.

6. A dynamic scheduling for online data integration is described based on the data behavior to tackle inefficiency of static scheduling of online integration plan. This approach is to minimize the number of data involved in a single operation by giving a higher priority to operations which potentially reduce the number of result data. Moreover, a scheduling algorithm is presented based on the sliding window model, and employs the statistic of data increment in the sliding window to determine the next online integration plan to be executed. This approach also minimize operations to update the materialized results which require expensive IO costs by deferring or terminating a plan.

7. The dynamic scheduling system is extended by processing of multiple data increments. The algebraic properties allow us to transform a data integration expression into a single increment expression for multiple data increments, and then produce one online integration plan. This approach reduces the operations to update the materialized result of computation and therefore increases the system performance.

8. The inefficiency of processing data integration on large size XML documents is tackled by processing them as fragmented XML documents. The XML data model in this thesis allows for the representation of incomplete XML documents. It is proposed to use one bounded and one rover data container to replace a data container in the data integration expression. This approach

adjusts the size of data increments to trigger their processing without waiting for the entire complete XML document to become available at the central site, therefore reducing waiting time to start processing a data increment. An additional algebraic operation for enabling computation of XML fragments is also presented. The algebraic properties allow the online processing of incomplete XML documents to achieve better performance of data integration on large XML documents.

1.3 Outline

The thesis document is organized as follows: Chapter 1 summarizes the motivation and problem statements. It also describes a brief idea on how to tackle the problem, and lists the thesis contributions.

Chapter 2 describes some works which are related to this thesis. It describes the most popular XML data models and XML algebras. Then, some existing incremental processing systems and algorithms are described. This is followed by a discussion of existing research in the stream processing on XML fragments. It includes a critical review of related works.

In Chapter 3 an XML data model and XML algebra are proposed to support online data integration system. Extended Tree Grammar is introduced as the data model of XML documents, and the operators to manipulate the structure of such documents are described. The XML algebraic operators which support online data integration systems are discussed, and the XML algebra properties which are important in the incremental processing system are described.

Chapter 4 describes the core of the thesis: online data integration systems. The processing of user queries from when they are received by the central site until results are sent back to user are discussed. A dynamic decomposition strategy is designed to balance processing between the central and remote sites. Then, the transformation of a data integration expression into an increment expression is described, followed by generation of an online integration plan. The dynamic scheduling system to achieve better performance on various data increment environments is also described. This chapter includes the algorithms and sufficient running examples in order to give a better explanation.

Next, in Chapter 5 the online data integration system is extended to work on concurrent data increments. A detail approach on re-optimization is described which reduces the high cost IO operations, and which processes multiple data increments at the same time. Modified algorithms of the online integration system

are described such that they are ready for processing concurrent data increments.

Chapter 6 extends the system to enable the processing of XML fragments. It describes the data model of a fragmented XML document and its components. It also includes the principles and assumptions made in order to process the fragments in a online integration system. Some Extended Tree Grammar's operators are appended to manipulate the structure of an XML document, as well as some XML algebraic operators.

Chapter 7 summarizes the study, draws conclusions, and lists some future avenues of research.

Chapter 2

Related Works

In this chapter a survey of some previous works which are related to this thesis are presented. In Section 2.1 some XML data models and algebras are reviewed, and existing approaches to formulate a data model and an XML algebra specifically for online integration of semistructured data are investigated. Section 2.2 describes several existing data integration systems. Then existing incremental query processing algorithms and data integration systems are reviewed in Section 2.3. Dynamic scheduling systems are discussed in Section 2.4, while in Section 2.5 some existing techniques to perform integration on XML fragments and XML stream data are discussed.

2.1 XML Data Model and Algebra

In the last decade, XML has became a ubiquitous standard for the representation of data, and hence the need for data integration of XML or semistructured data has emerged. Researchers have put a lot of their efforts into finding an appropriate formalization of a data model and algebra for the data integration of semistructured data. Generally, a schema is used to describe the structure of documents, and further to provide data constraints. Schema for an XML document can be classified into three groups based on their tree languages: local tree grammar (DTDs), single-type tree grammar (W3C XML-Schema) and regular tree grammar (RELAX Core, XDuce, TREX, RELAX NG) [38, 75, 78].

The evolution of a schema language for XML documents affected the query evolutions, and as a result various XML algebras have emerged. In general, an XML algebra is categorized into *tree-based* algebra and *tuple-based* algebra. YAT[24], XTasy[93], XAT[107], SAL[7] are examples of *tuple-based* algebras. On the other hand, TAX[57] and XAnswer[71] are classified as XML *tree-based* algebras. Meanwhile, DUMAX[20] provides *fuse node-based* and *tree-based* features.

2.1.1 XML Algebra (XAL)

In XAL (XML Algebra) [37], an XML document is modeled as a rooted connected directed cyclic or acyclic graph $G = (V, E, O, root)$, where V (vertices) represents elements or simple values, E is a set of directed edges to connect parent and child elements, O is a list contains edges which share the same parent and $root \in V$. Algebraic operators in XAL are classified into three groups of operator:

1. *Extraction operators*, which are used to retrieve relevant data from XML documents and to return a collection of vertices from an input XML document. It consists of *projection, selection, unorder, distinct, sort, join, cartesian product, union, difference*, and *intersection* operators.

2. *Meta-operators*, which provide a mechanism to express collections which appear more than once. The operators are used to express repetition at the input or operator level. Operators included in this group are *map*, and *Kleene Star* operators.

3. *Construction operators*, which are used to build an output XML document from data which are extracted from an input XML document. Construction operators include *create vertex, create edge*, and *copy examples* operators.

XAL operators are very similar to those of relational algebra but some operator definitions are not clear [17]. XAL is claimed to be more flexible than relational algebra, since the extraction operators work on collections with different types of elements [37].

2.1.2 XAnswer

XAnswer [71] defines its algebraic operations on a relational-like data structure. This algebra is based on some elements of XAT [108] and Galax [34]. Its algebraic operations are defined over ordered sets of tuples. XAnswer uses a data structure called Envelope ($\langle he|be|re\rangle$) where he represents header, be contains body and re is the result. A tuple in XAnswer is either a set or a sequence of single values [70].

XAnswer provides unary operators (*function execution, selection, projection, sort, index, nest, unnest, duplicate*) and binary operators (*union, cross product, left-outer-join*). XAnwer proposed some different operators to the relational algebraic operators. First, *union* operation in XAnswer does not remove duplicate tuples. Then, XAnswer introduces a new *left-outer-join* operator instead of expressing *left-outer-join* using *selection*, *cross product* and *union* operators.

XAnswer has the disadvantage of tuple-based algebras when transforming an XML document structure into tuples and vice versa by employing *nest* and *unnest* operators. Unlike other tuple-based algebras, *nest* and *unnest* operators in XAnswer are not complementary.

2.1.3 Tree Algebra for XML (TAX)

TAX (Tree algebra for XML) [57] is an XML algebra which represents a document in an ordered labeled tree. Every XML element is represented as a node which has:

1. A `tag` attribute which is a single-valued attribute to indicate the type of element;
2. A `content` attribute which represents atomic value and can be any of atomic types;
3. Several `pedigree` attributes to carry the information of element's predecessor. They are very useful for data manipulation and comparison.

TAX provides unary and binary operators (*Selection, Projection, Product, Grouping, Aggregation, Renaming, Reordering, Copy and Paste, Value Updates, Node Deletion, Node Insertion*) and some *set* operators (*intersection,difference*). TAX is a set-at-a-time algebra whose operators operate on one or more sets of XML documents and produce a set of XML documents as results [80, 82].

TAX defines a *pattern tree* to identify the subset of nodes of interest in any tree in a collection of tree, and to manipulate trees directly. A pattern tree can be used to bind a number of variables to represent multiple conditions in a single expression.

Despite the fact that most of TAX operators are compatible with the relational algebra, an extension of TAX includes two *join* operators [80] which are designed for different purposes. *ValueJoin* looks like an ordinary *join* operation which performs through application of a nested loop where a node value at the first set of XML documents matches with a node value at the second set of XML documents. *StructuralJoin* is an operation to connect two XML documents vertically, such that the second XML document becomes a sub-tree of the other. In this case, it is reasonable that TAX operates on an ordered set of XML document.

TAX has pros and cons [17, 93]. Its set of algebraic operators has clear semantics, and includes grouping and node deletion operators. On the other hand, the

concept of a *tree pattern* represents different concepts from the classical relational algebra. Carlo [93] criticizes that TAX optimization properties are not clear.

The idea of a *tree pattern* invites many researches to find its best performance in matching, reordering, expressiveness and optimization [45]. The importance of having a better algorithm to support *tree pattern* is shown by a number of *tree pattern* algorithms, such as: Generalized Tree Pattern [21, 22, 23], Annotated Tree Pattern [81, 83], Global Query Pattern Tree [105], and Twig Pattern [64, 102].

2.1.4 Discussion

The data model is an important factor because it determines the way we design a set of algebraic operators which are used to manipulate XML documents and their structures. Most existing XML data models can be classified into either a graph or a tree data model. A tree data model has better data structure than the graph for processing XML documents because both XML documents and memory have the same structures, a tree structure. The online data integration requires an XML data model which allows manipulation of XML documents in tree-based operations to obtain a better system performance.

On the other hand, data integration for a relational database has been established in a last decade. It incorporates numerous performance tuning algorithms which have been well proven. In order to utilize the performance tuning algorithms for the relational algebra, the online integration system requires a set of algebraic operations which are consistent with the relational algebra. Unfortunately, existing XML models whose algebraic operations are consistent with the relational algebra require expensive operation costs to convert the XML document tree structures into tuples and vice versa.

Furthermore, the existing XML algebras presented earlier have insufficient properties such that they can be used for online data integration systems. Therefore, the existing XML data models and their algebraic operations are used here as underlying support to propose an XML data model along with its algebraic operations to support online data integration system.

2.2 Data Integration System

The goal of a data integration system is to offer a uniform access to a set of autonomous and heterogeneous data sources. It deals with the task of combining

the contents of different information sources into global data. Data integration systems are generally equipped with a mediator at the central site and wrappers at the remote sites. A mediator has a role to provide a general view of data, to receive a user query, to send sub-queries to the wrappers, and then to integrate the sub-query results to produce a final answer to the user. Meanwhile, the wrappers have a responsibility to map the general view into the data sources.

Data integration systems can be classified based on how data access is controlled as follows [77, 91]:

1. In a *materialization approach*, data from participating remote sites are transformed into a local repository at the central site to be queried later. In this case, we have to discover all schema of the data sources at the remote sites. The global view of the network database is provided by a mediator. Moreover, wrappers have a role to transform the data sources into to a common data model. Then, the central site combines data received from the remote sites. In a data integration system, the user queries are answered by retrieval to the materialized database at the central site. This approach is widely used for Data Warehousing or Business Intelligence systems [60, 77, 90, 98, 103]

2. In a *virtualization approach*, information is accessed on-demand. Mediators have enough information about the availability of data sources and their wrappers. When a user query arrives at the central site, the mediator decomposes the user query into several sub-queries and sends them to the corresponding remote sites. In the next step, wrappers have a responsibility to send the results of the computation to the central site for further processing. In this approach, the central site does not store integrated information at a local repository [8, 27, 28, 65, 95].

A typical data integration solution needs to explore the following aspects [25]:

1. **Query system**: Since data integration systems work in a network database system, they require a query system which focuses on querying disparate data.

2. **Number of external data sources**: As the number of external databases increases, a more complex strategy is needed to obtain a better computation balance between the central site and external sites.

3. **Data source heterogeneity**: External data sources are often developed with various database systems, data models, and data structures. Data integration systems have to provide a method to transform the heterogeneous

structure of data sources into a uniform structure. They must have an algebraic model that operates on the designated uniform structure.

4. **External site autonomy**: The external sites often belong to disparate administrative entities, therefore the central site has limited access to data sources. The level of coordination between the central site and the external site must be maintained such that the central site has a correct global view of external sites.

Data integration systems based on the relational model have been widely researched for many years, which has led to numerous techniques to gather data sources from network database systems [46, 66, 67]. Achievements in the relational model integration are elaborated when XML becomes a standard for information interchange, but cannot be directly applied since their data structures are different.

One research direction in data integration is to provide an efficient query processing in a dynamic, scalable and heterogeneous network database environment [5, 15, 44, 55, 76]. Materialized views are often used to provide efficient decision-support queries [47, 50, 96, 104], but maintaining them is not an easy task. To obtain better performance, incremental processing is employed to maintain materialized views [11, 30].

Another research direction is to provide schema mapping between mediated and data sources [18, 58, 74, 84, 87, 106]. DIXSE [89] provides a data integration system to integrate *heterogeneous* data sources. It supports a semantic level integration, which takes several DTDs of the data sources and generate a semi-automatic conceptual schema. Almarimi [1] proposed a data integration framework which is able to resolve structural and semantic conflicts for distributed heterogeneous XML data. It provides a global XML schema as a homogeneous view of the heterogeneous data sources. Dong [26] proposed a probabilistic semantic mapping for uncertainty mapping.

Some researchers have proposed data integration system based on the similarity of data source's content and/or structure [6, 19, 33, 68, 85]. Viyanon [98] proposed an integration technique based on content and structure by detecting the similarity of subtrees. An XML data integration system based on an identification of nodes coming from different sources has been proposed in [86].

Data integration systems for semistructured data require a model and algebra that allow for an efficient processing of semistructured structures. The current author has proposed a *tree-based* XML algebra generalizing the relational algebra [48].

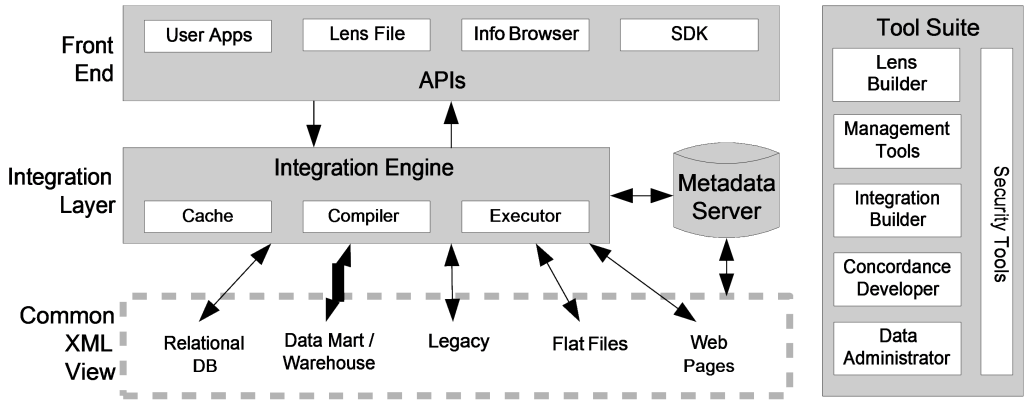

Figure 2.1: Nimble architecture (after [27, 28])

2.2.1 Logic-based XML Data Integration

Logic-based XML data integration was introduced by May [77] with an assumption that data sources at the remote sites do not change often. This data integration system employs a graph-based data model (XTreeGraph) to represent the overlapping of XML trees and XPathLog as a data manipulation and data integration language [77].

XTreeGraph is used as an internal data model not to represent an XML document, but its edge-labeled graph which represents a forest of overlapping XML trees to fit with data integration requirements. In this model, XML documents are defined as *views*, and subtrees may belong to several *tree views* [77].

This integration technique is classified as a semi-materializing approach, since not all objects are being copied and materialized at the local, integrated database. For some XML subtrees which are not structurally updated in the integration process, the system will preserve them at the original data sources and create a reference via links to them. Further, the system is willing to reuse the largest possible substructures of the original sources. User queries which are sent to the system are then forwarded to the references at runtime. This method will save *memory* and *time* to copy data, and ensures that user queries get the most up-to-date results.

2.2.2 Nimble XML Data Integration System

Nimble is a commercial data integration system which handles semistructured data and was designed based on the *integration engine* of XML-QL. It employs a set of *mediated schemas* in a *global-as-view* like approach [27]. Figure 2.1 shows the Nimble data integration system architecture.

Nimble provides a multi layer of access and multiple target sources. Its compiler translates and breaks a user query which is received at the integration engine into an appropriate query language [28]. Nimble tries to accommodate all types of queries and executes each of them according to the destination sources. Instead of employing a transformation procedure to convert data from the destination sources into a common data model, Nimble provides a universal algebra that supports operation on relational and semistructured data models.

2.3 Incremental Data Integration

In general, computation of a data integration system can be started whenever all data are available at the central site. *Online integration* is a process of continuous consolidation of data transmitted over wide area networks with data already available at a central site of a network database system [41]. It applies *online processing*, which means that theoretically infinite sequences of input data are processed in a *piece-by-piece* mode without having the entire set of data available from the very beginning [3, 35, 88, 100].

The continuity of the process requires the activation of a data integration procedure each time a new portion of data is received at a central site [41]. Online data integration systems do not delay the processing of incoming data until all transmissions from the remote sites are completed. Instead, the transmitted packets of data are integrated with the partial results as soon as they arrive at a central site. Such an approach reduces time spent by a user in waiting for the first result from a running application, and it allows for an early termination of an application when the initial results are inconsistent with expectations.

An efficient implementation of online algorithms is based on the principle of *incremental* and/or *decremental* processing of data, where the current state of processing is combined with the increments and/or decrements of incoming data in order to obtain a new state of processing.

An important advantage of online integration is its ability to utilize unused computational resources at a central site while data is transmitted over a network. Foster [36] concluded that freely available distributed data sets and fast wide-area networks will promote online data integration as an important research area of distributed computing. The performance of online data integration depends on the advanced online algorithms used for consolidation of data, and the efficient optimization of online integration plans. Getta [39], proposed an optimization of task processing schedules where a task submitted at a central site is decomposed

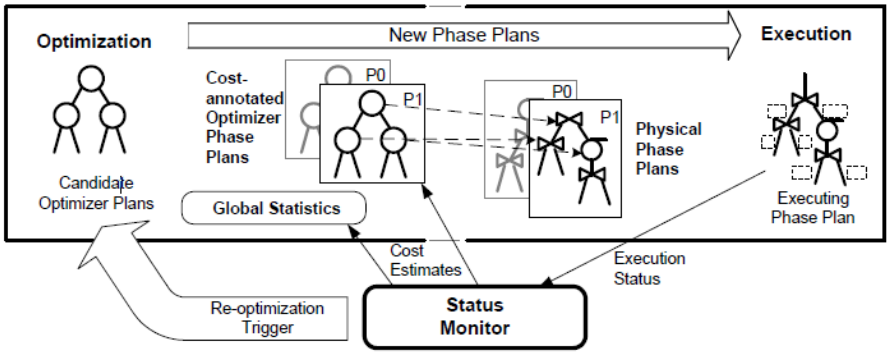

Figure 2.2: Tukwila query processor (after [55])

into a number of individual tasks for processing at the remote sites. He proposed a number of static optimization techniques for data integration plans in a relational algebra expression.

The research on online integration can be traced back to the work on query processing in a network or federated database systems [41, 94]. Unpredictable behavior of the data transmissions in wide area networks and strong autonomy of the remote database systems makes the estimation of query processing time difficult and imprecise. A query processing plan could be optimal only when it matches a certain circumstance, but falls to its worst on others. Therefore, a reactive execution plan is needed in order to adapt to network circumstance and remote site availability.

According to Ives [55], the following techniques can be employed in order to get a good performance of online integration on semistructured data:

1. equip the system with a pipelined execution for streaming XML data;
2. using adaptive operators for processing over various data transfer rates; and
3. query re-optimization.

Ives proposed Tukwila [55] which presents an adaptive optimization query processor, as shown in Figure 2.2. Sayed [29] proposed a system to maintain materialized XQuery view by performing incremental update to gain better access to data sources. Fegaras [30] and Bonifati [11] proposed systems for incremental maintenance of the XML view. Salem in [91] proposed an integration system for near real-time requirements and realized data which utilizes Active XML.

2.3.1 Incremental Maintenance of Materialized XML Views

One of the biggest challenges in a *materialization approach* to data integration is maintaining a materialized database with up-to-date data sources. Whenever an update occurs at a data source, the corresponding materialized view at the integration database must be updated such that the user query to the integration database gets the most up-to-date data. Maintaining materialized views by execution of the corresponding views over updated data sources requires re-computation cost which is too expensive. Incremental processing to maintain a materialized XML view provides freshness of views of data with less operation costs [2, 30].

Abiteboul [2] proposed an algorithm which produces a set of queries when the source has different states. Then, the queries are executed over the XML view to obtain the most up-to-date results. The system is based on a graph data model, and a query language Lorel. The algorithm is scalable and has a simple query execution strategy, but in some situations the incremental maintenance performance can be as good as full re-computation of XML view.

Fegaras [30] generates a view XQuery expression e' which is a *right-inverse* of a view XQuery expression e such that $e(e'(V)) = V$ where V is a state of view. Then, a composition $F(V)$ is generated such that $F(V) = e(u(e'(V)))$ and u represent update to the source data. In the next step, $F(V)$ is transformed into a set of XQuery updates (XUF) which modify an XML view V to reach a new state of XML view.

2.3.2 Incremental Recomputations in Materialized Data Integration

Computation of *materialized view*, *materialized data integration* and MapReduce share a similar process. They take data from heterogeneous data sources, perform some transformation techniques, and store the results back to a database system. A materialized view uses a user-defined view definition to perform the transformation. Meanwhile, a materialized data integration employs the ETL process (Extract, Transform and Load), and MapReduce employs a user-defined map and reduce functions [60, 61].

Although these algorithms are similar, there is an important difference in the technique used to get a new state of results whenever the data sources are updated. A materialized view employs a user function to enable incremental computation such that an incoming update is computed and merged to the current results without re-computation of the data sources from scratch. However, incremental

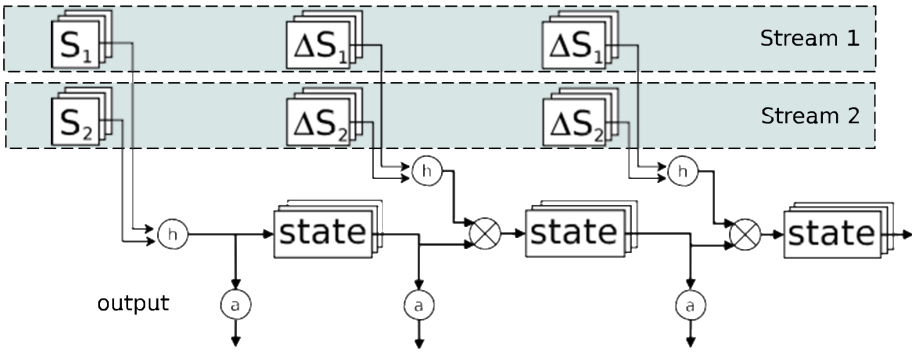

Figure 2.3: Incremental query processing on Big Data streams (after [31])

computation is not supported in materialized data integration and MapReduce, and therefore re-computation must be performed to get the most up-to-date results [60, 61].

Jörg [60] proposed an incremental processing approach to materialized data integration and MapReduce using incremental techniques in the maintenance view. Incremental processing on materialized data integration is based on algebraic differencing of ETL jobs using recursive applications of standard delta rules [43] to the original algebra expression. Furthermore, it uses Magic Sets to reduce the CPU cost as cleansing overhead of ETL incremental processing.

2.3.3 Incremental Query Processing on Big Data Streams

Recently, Fegaras [31] proposed an incremental query processing for a large-scale database called MRQL Streaming. This incremental processing works on Big Data streams by transforming any SQL-like query into an incremental distributed stream processing engine (DPSE) program to produce accurate results, not approximate answers. Analysis tools based on batch processing may be expensive to compute Big Data that grows rapidly.

In MRQL Streaming, a dataset is defined as a set of hierarchical data like XML or JSON. A streaming query is expressed as $q(\overline{S})$ where $S_i \in \overline{S}$ and $S_i : i = 0, \ldots, n$ is a streaming data source. S_i contains an initial dataset and is followed by a continuous incremental stream ΔS_i in time interval Δt.

For a *monoid homomorphic* query $h(\overline{S})$, incremental processing is performed by combining the result query at time t with the results of a query on ΔS_i, such that $h(\overline{S \uplus \Delta S}) = h(\overline{S}) \bigotimes h(\overline{\Delta S})$. $\bigotimes$ is a merge function which is implemented as a partitioned *join*. Meanwhile, a *non-monoid homomorphic* query $q(\overline{S})$ is transformed such that all non-monoid homomorphic parts are pulled out and a monoid homomorphic query h is constructed, $q(\overline{S}) = a(h(\overline{S}))$.

In the case of a streaming dataset which contains insertions and deletions, Fegaras [31] creates a decremental batch $\Delta S_i'$ such that the dataset becomes $S_i \uplus \Delta S_i - \Delta S_i'$ at time $t + \Delta t$. In this research, $\Delta S_i'$ is required to be available in the stream ($\Delta S_i \subseteq S_i$). This means that the decremental updates have to be placed at the end of the stream in order to ensure that the system generates the correct results.

2.4 Dynamic Query Scheduling Systems

In order to compute a user request, a query execution engine transforms a user query into an ordered physical algebraic operation called *query execution plan*. A static scheduling system allows sequential processing of a QEP, starting from the first algebraic operation until all operations are executed. A static scheduling system is good for simple QEPs, which do not require massive I/O access, and has minimal idle CPU cycles. In a single processor system, optimization of query processing can be obtained by modification of QEP executions such that the problem of a static scheduling system can be overcome.

Query scrambling is a popular dynamic scheduling strategy in a network database system. It modifies the query plan whenever unexpected delay, at initial delay, burst arrival, and/or slow delivery, occurs at any data source. The dynamic scheduling strategy proposed in [4, 97] optimizes query execution by reducing idle time. Ives [55, 56] proposed overlapping multiple I/O operations from different data sources and pipelined execution to utilize idle time and achieve faster computation.

An efficient dynamic scheduling system must be supported by a monitoring system which will continuously collect behaviour of the query engine in order to find the optimal execution time. Gounaris [42] proposed self-monitoring query operators to obtain completion time and number of results. The information is useful to refine the cost model and to decide the next query plan to execute in order to obtain optimal performance.

Bouganim [14] proposed an integration system which includes delay in the execution strategy by monitoring the arrival rate and available memory. Meanwhile, Getta [40] proposed a combination of query scrambling and reduction technique for integration system.

2.5 Processing on Incomplete XML Documents

Incomplete semistructure data refers to a case where some parts of an XML document are not available at the central site before integration starts. Incompleteness of a document can be as follows [59]:

1. **Structural incompleteness**, where some parts of the structure of a document have not been received at the central site. Structural incompleteness always leads to intractability of query answering.

2. **Labeling and data value incompleteness**, where the structure of a document is available at the central site although some labellings and data values are not.

A set of XML fragments which are available at the central site forms incomplete documents. In some cases, incomplete documents have enough properties to allow computation without having the rest fragments arrive at the central site.

XML fragmentation is often used to increase the performance of query processing by cutting queries into smaller sub-queries to operate concurrently on fragments, then the results of sub-queries are combined to obtain the final results.

2.5.1 Fragmentation Techniques

XML fragmentation is divided into two major techniques, *ad-hoc* and *structured* fragmentation [79]. *Ad-hoc* fragmentation is a technique where arbitrarily nodes are removed from the origin of the XML document. On the other hand, *structured* fragmentation is a technique based on the defined schema. XML fragments in *structured* fragmentation are usually generated by application of a set of algebraic operations [16].

Constraint-based fragmentation can be classified as *ad-hoc* one, where the fragmentation is obtained according to specific properties of the fragments. Bonifati [10] proposed the SimpleX algorithm to create XML vertical fragmentation with a structural constraint on size, tree-width, and tree-depth. SimpleX has a set of top-down heuristics which start from the root of XML documents and cut the sub-tree whenever it satisfies the constraint of size, tree-width and tree-depth. Conversely, Jin [59] proposed a *cost-based* fragmentation where the *upper size limit* of an XML fragment becomes an important parameter.

Hole-filler, introduced by Bose [13], is an *ad-hoc* fragmentation technique which has attracted many researchers. In the *hole-filler* model, a complete XML

document can be pruned arbitrarily into a number of XML fragments. Every XML fragment is associated with a unique *filler* ID and has a set of *holes* which represent empty places where other fragments could be connected to form a complete document. The structure of the original XML document and how it is fragmented is presented in a simple recursive DTD named *Tag Structure* [13].

Lee [92] proposed **XML fragment labeling (XFL)** to improve *hole-filler* with *Dewey order encoding* as a labeling system for XML fragments. XFL enables processing of dynamic stream size and is equipped with *XPath step reduction* for query processing optimation.

Structured fragmentation, on the other hand, takes advantage of fragmentation in the relational model. Most of the techniques are related to the problem of how to break a large amount of XML databases into a number of physical storages. Braganholo [16] classifies XML fragmentation techniques as: *horizontal*, *vertical*, and *hybrid* fragmentation. Horizontal fragmentation divides a set of XML documents into some sets of XML documents according to their matching criteria. Horizontal fragmentation is constructed by application of *selection* over a set of XML documents, therefore their schema remains the same. By contrast, in the vertical fragmentation, an XML document in a source database is cut into some smaller XML fragments by application of the *projection* operation, then XML documents which have the same schema are located at a set. As a result, the XML fragments have different schema than its origin document. Hybrid fragmentation is a composition of horizontal and vertical fragmentation [16]. Ma [72] proposed similar fragmentation techniques, but using an object database approach to define horizontal, vertical and split fragmentations.

Birhani [9] proposed two fragmentation models, *query based* and *structure-and-size based* fragmentation. In a query based fragmentation, a vertical fragmentation is based on bond energy and graphical based algorithm where an XML document is projected according to groups of its elements. Meanwhile, *structured-and-size-based* fragmentation is proposed to partition XML documents according to the limitation of the device which will use the data. However, these two algorithms are not formally defined in his paper.

2.5.2 Query Processing on XML Stream Data

XML stream data is a form of data source which continuously arrive at the system. They are typically small, unordered and may be unbounded. Evaluation on XML stream data is challenging because it requires an algorithm which processes the

stream in one scan and returns the partial result. Most of query processing techniques on XML stream data are based on XQuery and XPath streaming evaluation [101].

TurboXPath [62] is one of the XML streaming evaluations on XPath. It is based on tree-pattern query and employs an array data structure to record evaluation status [101]. TurboXPath produces a sequence of XML fragment tuples which are constructed by computation of all candidate answers using a nested-loop join algorithm. TurboXPath performance falls to its worst case for processing query on recursive XML documents.

Processing XML stream data using XQuery evaluation utilizes XQuery which is more expressive than XPath. Some XML stream processing which are based on XQuery evaluation refer to a fragmented XML stream in the concept of *hole-filler* [69, 12, 13, 99, 52]. In addition to *hole-filler* model, Fegaras and Bose in [32, 13] proposed a query algebra for XQuery which works for XML stream data. The query is based on the nested relational algebra and consists of *extraction, selection, merge, join, reduce, nest,* and *unnest* operators. This query algebra is designed to perform pipelining of the algebraic operators in main memory.

Huo [54] proposed XFPro, a framework and algorithm to compute XPath queries on a stream of XML fragments. XFPro utilizes *hole-filler* to model XML fragment and employs a set of transformations to convert an XPath query expression on a complete XML document into an optimized query plan. Huo [53] enhanced the *hole-filler* method by utilization of query statistics. It has two query statistics, *path frequency tree* is used to increase utilization of fragment, and *Markov tables* are used in order to increase query performance. He also presented a cost model for fragmented XML stream processing [53].

Meanwhile, Koch [63] proposed XML Stream Attribute Grammars (XSAGs) for scalable query processing on XML streams. An XSAG is based on an *extended regular tree grammar* which is annotated with attribute functions to specify the output to be produced from the input stream. In this context, XSAG allows actual data transformation from an input stream, instead of just document filtering. An XSAG is translated into a deterministic pushdown transducer (DPDT) which ensures that the size of memory used remains proportional to the depth of an XML document, and therefore guarantees that queries to an input XML stream can be evaluated in linear time.

However, XSAG does not consider the element's attribute of XML document in its definition, and hence evaluation on XML streams is mainly based on the matching value of node elements. Furthermore, binary operations on XML streams are not described in [63], whereas having binary operations is an important feature in data integration systems.

The related works discussed in this chapter provide important underlying justification to design an appropriate XML data model along with its algebraic operations, and to propose an online algorithm for data integration system which is suitable for scalable XML documents in a multi-database network system.

Chapter 3

XML Data Model and Algebra

In this chapter an XML data model and algebra is presented as an underlying architecture for an online integration system. Section 3.1 describes some XML documents which will be used as examples to give clearer explanations. Section 3.2 describes an XML data model, and Section 3.3 describes operations on Extended Tree Grammar. In Section 3.4 XML algebra operators are introduced, based on the XML data model designed, and in Section 3.5 XML algebra rules and properties to support online processing are described. Section 3.6 presents a brief comparison between the proposed XML algebra and relational algebra.

3.1 XML Document Working Examples

The following XML documents (Figures 3.1 through 3.3) are presented in order to provide a better explanation on how the XML data model and XML algebra operators are designed. The XML document in Figure 3.1 contains information about `book`s. Every `book` has a `title`, a set of author's email address in the `authors` element, an alternative element either `subject` or `genre`, and zero or more editor's email addresses. A book element has at least an author email address since a book must be written by at least one author. Meanwhile, the XML document in Figure 3.2 contains data about `author`s which includes an author `name` and his/her `email` address. The `editor` data in Figure 3.3 includes a `name` and an `email` address element.

In a data integration system, we assume that XML documents to be processed reside at different sites such that a global uniform schema cannot be provided. For example, in the `book` document, an editor's email is stored in an element named `ed_email`, but in the `editor` document, it is stored in an element named `email`.

It is assumed that yhe node elements in XML documents do not contain mixed content, and we consider that an XML document may use duplicate element names within the document. For example, after processing an algebraic operation against

```
<book>
   <title>XML</title>
   <authors>
      <aut_email>andy@yahoo.com</aut_email>
      <aut_email>ben@yahoo.com</aut_email>
   </authors>
   <subject>Data</subject>
   <ed_email>ross@gmail.com</ed_email>
</book>

<book>
   <title>Harry Potter and the Philosopher's Stone</title>
   <authors>
      <aut_email>jk@yahoo.com</aut_email>
   </authors>
   <genre>Fantasy</genre>
</book>
```

Figure 3.1: Two instances of `book` XML document

`author` and `editor` documents, we may have two `email` elements in the result document. In the relational model, a new field name is assigned to each instance of the duplicate elements such that the table contains unique field names.

```
<author>
  <name>Andy Cole</name>
  <email>andy@yahoo.com</email>
</author>

<author>
  <name>Ben Johnson</name>
  <email>ben@yahoo.com</email>
</author>

<author>
   <name>J.K. Rowling</name>
   <email>jk@yahoo.com</email>
</author>
```

Figure 3.2: Three instances of `author` XML document

```
<editor>
   <name>Ross Marrie</name>
   <email>ross@gmail.com</email>
</editor>
```

Figure 3.3: An `editor` XML document

3.2 XML Data Model for Online Processing

An online data integration system for semistructured data requires a data model which allows utilization of the performance tuning algorithms in the relational model. The data model has to eliminate the expensive cost of *nest* and *unnest* operations to transform the document tree structures into tuples and vice versa. Among all the existing data models for XML documents, the tree based data model meets the needs of having easy operations to modify the structure of XML documents. Furthermore, the theory of *regular tree grammar* has been used in various aspects of XML schema language and XML document query processing.

We use the concept of regular tree grammar introduced in [63, 78] to define a data model for XML documents. A data model for XML documents which contain only elements without any attribute can be formally defined by a *Regular Tree Grammar* (RTG) [51].

Definition 1. *Let N be a set of non-terminal symbols, and ϵ be an empty symbol. A regular expression r over non-terminal symbols $N \cup \{\epsilon\}$ is defined as follows:*

1. *ϵ is a regular expression.*
2. *for each $X \in N$, X is a regular expression.*
3. *if r and s are regular expressions, then $r|s$, r_s, $r+$, $r*$, $r?$, and (r) are regular expressions.*

Definition 2. *Let r be a regular expression over $N \cup \{\epsilon\}$. A Regular Tree Grammar is a context free grammar defined as a 4-tuple $G = (N, T, S, P)$, where N is a finite set of non-terminal symbols, T is a finite set of terminal symbols, and $S \in N$ is a start symbol. P is a set of production rules in the form of $\mathtt{X} \rightarrow \mathtt{t}(r)$ where $\mathtt{X} \in N$, $\mathtt{t} \in T$.*

A production rule with $r = \epsilon$ can be written as $\mathtt{X} \rightarrow \mathtt{t}(\epsilon)$, and is equivalent to a production rule $\mathtt{X} \rightarrow \mathtt{t}$ as $\mathtt{t}(\epsilon) \equiv \mathtt{t}$. Examples 3.1, 3.2 and 3.3 show Regular Tree Grammars for XML documents of `BOOK`, `AUTHOR` and `EDITOR`, respectively.

Example 3.1. *A Regular Tree Grammar for* `BOOK` data.

```
N={S,TITLE,AUTHORS,AUT_EMAIL,SUBJECT,GENRE,ED_EMAIL}
T={book,title,authors,aut_email,subject,genre,ed_email}
P={S→book(TITLE AUTHORS (SUBJECT|GENRE) ED_EMAIL?),TITLE→title,
   AUTHORS→authors(AUT_EMAIL+),SUBJECT→subject,GENRE→genre,
   ED_EMAIL→ed_email,AUT_EMAIL→aut_email}
```

□

Example 3.2. *A Regular Tree Grammar for* `AUTHOR` data.

```
N={S,NAME,EMAIL}
T={author,name,email}
P={S→author(NAME EMAIL),NAME→name,EMAIL→email}
```

□

Example 3.3. *A Regular Tree Grammar for* `EDITOR` data.

```
N={S,NAME,EMAIL}
T={editor,name,email}
P={S→editor(NAME EMAIL),NAME→name,EMAIL→email}
```

□

An XML model based on *Regular Tree Grammar* allows us to create a structure for XML tree documents with "infinite arity", where a particular element may be used in multiple levels of an XML document. Example 3.4 shows a grammar for an XML document with infinite arities to represent a multi-level marketing data where a `member` can have some other members as his downline.

Example 3.4. *A Regular Tree Grammar for* `multi-level marketing` data.

```
N={S,MEMBER}
T={mlm,member}
P={S→mlm(MEMBER),MEMBER→member(MEMBER),MEMBER→member}
```

□

Definition 3. *Let $G = (N, T, S, P)$ be a Regular Tree Grammar. A string is a sequence of elements from T and N, and denoted as $(T \cup N)^*$.*

Definition 4. *Let $G = (N, T, S, P)$ be a Regular Tree Grammar, v and w be strings. Let r, s be regular expressions over $N \cup \{\emptyset\}$ in v. w is derivable from v, denoted as $v \Rightarrow_G w$ if w can be constructed by application of any production rule in G to a string v. Before application of a production rule, all regular expression operators $(|, *, +, ?)$ in v are translated such that v contains no operators, as the following:*

1. $(\boldsymbol{r}|\boldsymbol{s}) \rightarrow \boldsymbol{r}$ *or* $\boldsymbol{s}$
2. $\boldsymbol{r}* \rightarrow \epsilon$ *or* $\boldsymbol{r}$ *or* $\boldsymbol{r}\sqcup\boldsymbol{r}$ *or* $\ldots$ *or* $\boldsymbol{r}\sqcup\boldsymbol{r}\ldots\sqcup\boldsymbol{r}$
3. $\boldsymbol{r}+ \rightarrow \boldsymbol{r}\sqcup\boldsymbol{r}*$
4. $\boldsymbol{r}? \rightarrow \epsilon$ *or* $\boldsymbol{r}$

Notation $\Rightarrow_G*$ is used to represent multiple derivations of production rules in G.

Definition 5. *Let $G = (N, T, S, P)$ be a Regular Tree Grammar.*

1. *$w \in (T \cup N)^*$ is a sentence form of G if there is a derivation $S \Rightarrow_G * w$ in G.*
2. *$w \in T^*$ is a sentence of G if there is a derivation $S \Rightarrow_G * w$ in G. A sentence is a string over terminal symbols which is derived from a start symbol.*
3. *The language of G, denoted $L(G)$, is the set $\{w \in T* \mid S \Rightarrow_G * w\}$*

Example 3.5. *Derivation of a Regular Tree Grammar for* `book` *structure as in Example 3.1.*

```
S⇒book(TITLE AUTHORS (SUBJECT|GENRE) ED_EMAIL?)
 ⇒book(title AUTHORS SUBJECT ED_EMAIL)
 ⇒book(title authors(AUT_EMAIL+) SUBJECT ED_EMAIL)
 ⇒book(title authors(aut_email AUT_EMAIL) SUBJECT ED_EMAIL)
 ⇒book(title authors(aut_email aut_email) SUBJECT ED_EMAIL)
 ⇒book(title authors(aut_email aut_email) subject ED_EMAIL)
 ⇒book(title authors(aut_email aut_email) subject ed_email)
```

Further, we can derive a different sentence from RTG in Example 3.1 such as:
`book(title authors(aut_email) genre)` □

Definition 6. *Let G=(N,T,S,P) be a Regular Tree Grammar, m be a sentence in $L(G)$, and $t \in T$ be a terminal symbol. n is a subtree of m if there exists a terminal symbol t such that $m = t(n)$.*

Definition 7. *Let G be a Regular Tree Grammar, m be a sentence in $L(G)$, n be a subtree of m, and $t \in T$ is a terminal symbol. An attribute-free XML document with a structure m, denoted as $x(m)$ is defined as a result of transformation (text substitution) of m such that:*

1. *when m has a subtree n, $m = t(n)$:* `t(n)`→`<t>n</t>`
2. *when $n{=}\epsilon$, $m = t(\epsilon) = t$:* `t`→`<t>#PCDATA</t>`

where `#PCDATA` *(Parsed Character Data) are characters which can be parsed by the XML parser, and $\epsilon \in$*`#PCDATA`*.*

Example 3.6. *Transformation of a sentence of Regular Tree Grammar in Example 3.1:* $m{=}$ `book(title authors(aut_email aut_email) subject ed_email)`. Transformation is performed in pre-order traversal, starting from the left most terminal symbol of the sentence to the right. The transformation is shown step by step as follows:

a. The first symbol to transform is a `book` terminal symbol. Transformation of `book` terminal symbol is as follows:
`book(title authors(aut_email aut_email) subject ed_email)`→

```
<book>
   title authors(aut_email aut_email) subject ed_email
</book>
```

b. The next terminal symbol to transform is `title`. The transformation of `title` terminal symbol is as follows:
`title authors(aut_email aut_email) subject ed_email` →

```
<book>
   <title>XML Bible</title>
   authors(aut_email aut_email) subject ed_email
</book>
```

c. Then, transformation of `authors` terminal symbol is as follows:
`authors(aut_email aut_email) subject ed_email` →

```
<book>
   <title>XML Bible</title>
   <authors>
      aut_email aut_email
   </authors>
   subject ed_email
</book>
```

d. Transformation of the first `aut_email` terminal symbol is as follows:
`aut_email aut_email` →

```
<book>
   <title>XML Bible</title>
   <authors>
      <aut_email>andy@yahoo.com</aut_email>
      aut_email
   </authors>
   subject ed_email
</book>
```

e. Transformation of the second `aut_email` terminal symbol is:
`aut_email` →

```
<book>
   <title>XML Bible</title>
   <authors>
      <aut_email>andy@yahoo.com</aut_email>
      <aut_email>ben@yahoo.com</aut_email>
   </authors>
   subject ed_email
</book>
```

f. Transformation of `subject` terminal symbol is as follows:
`subject ed_email` →

```
<book>
   <title>XML Bible</title>
   <authors>
      <aut_email>andy@yahoo.com</aut_email>
      <aut_email>ben@yahoo.com</aut_email>
   </authors>
   <subject>Database</subject>
   ed_email
</book>
```

g. Transformation of `ed_email` terminal symbol is as follows:
`ed_email` →

```
<book>
   <title>XML Bible</title>
   <authors>
      <aut_email>andy@yahoo.com</aut_email>
      <aut_email>ben@yahoo.com</aut_email>
   </authors>
   <subject>Database</subject>
   <ed_email>ross@gmail.com</ed_email>
</book>
```

□

Online processing of semi-structured data requires every XML document to have a unique immutable identifier. For this purpose, an XML document is encapsulated by an additional parent element with an attribute to store the identity. The XML documents in Figure 3.1, 3.2 and 3.3 are modified to include identifiers as in Figures 3.4, 3.5 and 3.6.

```
<xml id="B01">
   <book isbn="9872347765" lang="EN">
      <title>XML</title>
      <authors>
         <aut_email>andy@yahoo.com</aut_email>
         <aut_email>ben@yahoo.com</aut_email>
      </authors>
      <subject>Data</subject>
      <ed_email>ross@gmail.com</ed_email>
   </book>
</xml>

<xml id="B02">
   <book isbn="9788700631625" lang="EN">
      <title>Harry Potter and the Philosopher's Stone</title>
      <authors>
         <aut_email>jk@yahoo.com</aut_email>
      </authors>
      <genre>Fantasy</genre>
   </book>
</xml>
```

Figure 3.4: XML documents for `book` data with unique IDs

Regular Tree Grammar defined in [78] does not include attribute definitions. Then, our idea is to extend the definition of regular tree grammar with an attribute definition.

Definition 8. *Let r be a regular expression over non-terminal symbols $N \cup \{\epsilon\}$. An Extended Tree Grammar (ETG) is a 5-tuple $G = (N, T, A, S, P)$, where N is a finite set of non-terminal symbols, T is a finite set of terminal symbols, A is a finite set of attribute symbols, and $S \in N$ is a start symbol. P is a set of production rules in a form of* `X→t[A'](r)` *where* `X`$\in N$, `t`$\in T$, `A'`$\subseteq A$, *and attributes may show up in any order. P includes exactly one production rule for start symbol* `S→xml[id](r)`.

The RTG for the `BOOK` document in Example 3.1 is extended into an ETG to

```
<xml id="A01">
   <author>
      <name>Andy Cole</name>
      <email>andy@yahoo.com</email>
   </author>
<xml>

<xml id="A02">
   <author>
      <name>Ben Johnson</name>
      <email>ben@yahoo.com</email>
   </author>
<xml>

<xml id="A03">
   <author>
      <name>J.K. Rowling</name>
      <email>jk@yahoo.com</email>
   </author>
</xml>
```

Figure 3.5: XML documents for `author` data with unique IDs

```
<xml id="E01">
   <editor>
      <name>Ross Marrie</name>
      <email>ross@gmail.com</email>
   </editor>
<xml>
```

Figure 3.6: An XML document for `editor` data with a unique ID

include `id`, `isbn` and `lang` attribute symbols (see Example 3.7). The production rule for the start symbol must be modified such that it represents a document is started with an `xml` element with a unique `id` attribute. Then, for the XML document in Figure 3.4, the production rule of the start symbol is `S→xml[id](BOOK)`.

Further, since the document in Figure 3.4 has two additional attributes for `book` element (ISBN and language), the production rule for the `BOOK` element is modified into `BOOK→book[isbn lang](TITLE AUTHORS SUBJECT ED_EMAIL?)`. The complete ETG for XML documents `book`, `author`, and `editor` are shown in Example 3.7, 3.8, 3.9.

Example 3.7. *An Extended Tree Grammar for* `BOOK` data.

`N={S,BOOK,TITLE,AUTHORS,AUT_EMAIL,SUBJECT,GENRE,ED_EMAIL}`

T={xml,book,title,authors,aut_email,subject,genre,ed_email}
A={id,isbn,lang}
P={S→xml[id](BOOK),BOOK→book[isbn lang](TITLE AUTHORS (SUBJECT|GENRE)
ED_EMAIL?),TITLE→title,AUTHORS→authors(AUT_EMAIL+),SUBJECT→subject,
GENRE→genre,AUT_EMAIL→aut_email,ED_EMAIL→ed_email}

□

Example 3.8. *An Extended Tree Grammar for an* AUTHOR data.

N={S,AUTHOR,NAME,EMAIL}
T={xml,author,name,email}
A={id}
P={S→xml[id](AUTHOR),AUTHOR→author(NAME EMAIL),
NAME→name,EMAIL→email}

□

Example 3.9. *An Extended Tree Grammar for an* EDITOR data.

N={S,EDITOR,NAME,EMAIL}
T={xml,editor,name,email}
A={id}
P={S→xml[id](EDITOR),EDITOR→editor(NAME EMAIL),
NAME→name,EMAIL→email}

□

Example 3.10. *Derivation of a sentence in ETG of* book *document G=(N,T,A,S,P)*
Based on an ETG in Example 3.7, we are able to derive a sentence as follows:

S⇒xml[id](BOOK)
⇒xml[id](book[isbn lang](TITLE AUTHORS (SUBJECT|GENRE) ED_EMAIL?))
⇒xml[id](book[isbn lang](title AUTHORS SUBJECT ED_EMAIL))
⇒xml[id](book[isbn lang](title authors(AUT_EMAIL+) SUBJECT ED_EMAIL))
⇒xml[id](book[isbn lang](title authors(aut_email AUT_EMAIL) SUBJECT ED_EMAIL))
⇒xml[id](book[isbn lang](title authors(aut_email aut_email) SUBJECT ED_EMAIL))
⇒xml[id](book[isbn lang](title authors(aut_email aut_email) subject ED_EMAIL))
⇒xml[id](book[isbn lang](title authors(aut_email aut_email) subject ed_email))

Moreover, we obtain a different sentence from ETG in Example 3.7 such that it has one aut_email element, a genre instead of a subject element and no ed_email element as the following:
xml[id](book(title[isbn lang] authors(aut_email) genre)). □

The sentence created in Example 3.10 is then transformed into an instance of the XML document.

Definition 9. *Let G=(N,T,A,S,P) be an Extended Tree Grammar, m be a sentence in L(G), and $t \in T$ be a terminal symbol. n is a subtree of m if there exists a terminal symbol t such that* `m=t[A'](n)`.

A sentence of an ETG is defined as follows:

Definition 10. *Let G=(N,T,A,S,P) be an Extended Tree Grammar.*

1. *$w \in (T \cup N \cup A)*$ is a sentence form of G if there is a derivation $S \Rightarrow_G * w$ in G.*
2. *$w \in (T \cup A)*$ is a sentence of G if there is a derivation $S \Rightarrow_G * w$ in G. A sentence is a string over terminal symbols and attribute symbols which is derived from a start symbol.*
3. *The language of G, denoted L(G), is the set $\{w \in (T \cup A)* \mid S \Rightarrow_G * w\}$*

Based on the created ETG, an XML document can be re-defined as in Definition 11 to cover instantiation of an attribute.

Definition 11. *Let G=(N,T,A,S,P) be an Extended Tree Grammar, m be a sentence in G, and n be a subtree of m. Let $t \in T$ be a terminal symbol, and $a_1, \ldots a_n \in A$ be attribute symbols. An XML document with a structure m is denoted as $x(m)$ and is defined as a result of transformation (text substitution) of m such that:*

1. *when m has a subtree n, then m=t(n):*
   `t[`$a_1, \ldots, a_n$`](n)`$\rightarrow$`<t` a_1`="#PCDATA" ...` a_n`="#PCDATA">`n`</t>`
2. *when $n=\epsilon$, then $m=t(\epsilon)=t$:*
   `t[`$a_1, \ldots, a_n$`]`$\rightarrow$`<t` a_1`="#PCDATA" ...` a_n`="#PCDATA">#PCDATA</t>`

$x(m)$ is assigned to a unique immutable identity which is stored in an `id` *attribute of element* `xml`.

Example 3.11. *Transformation of a sentence of ETG in Example 3.7:*
`xml[id](book[isbn lang](title authors(aut_email aut_email) subject ed_email))`.
Transformation is performed in pre-order traversal, starting from the left most terminal symbol to the right, and is shown step by step as follows:

a. First, transformation of `xml[id]` terminal symbol from a sentence: `xml[id](book [isbn lang](title authors(aut_email aut_email) subject ed_email))`$\rightarrow$

```
<xml id="B01">
   book(title authors(aut_email aut_email) subject ed_email)
</xml>
```

b. Next, transformation of `book[isbn lang]` terminal symbol from: `book[isbn lang] (title authors(aut_email aut_email) subject ed_email)`→

```
<xml id="B01">
   <book isbn="9872347765" lang="EN">
      title authors(aut_email aut_email) subject ed_email
   </book>
</xml>
```

c. Then, transformation of `title` terminal symbol from:
`title authors(aut_email aut_email) subject ed_email` →

```
<xml id="B01">
   <book isbn="9872347765" lang="EN">
      <title>XML Bible</title>
      authors(aut_email aut_email) subject ed_email
   </book>
</xml>
```

d. Transformation of `authors` terminal symbol is as follows:
`authors(aut_email aut_email) subject` →

```
<xml id="B01">
   <book isbn="9872347765" lang="EN">
      <title>XML Bible</title>
      <authors>
         aut_email aut_email
      </authors>
      subject ed_email
   </book>
</xml>
```

e. Transformation of the first `aut_email` terminal symbol is as follows:
`aut_email aut_email` →

```
<xml id="B01">
   <book isbn="9872347765" lang="EN">
      <title>XML Bible</title>
      <authors>
         <aut_email>andy@yahoo.com</aut_email>
         aut_email
      </authors>
      subject ed_email
   </book>
</xml>
```

f. Transformation of the second `aut_email` terminal symbol is as follows:
`aut_email` →

```
<xml id="B01">
   <book isbn="9872347765" lang="EN">
      <title>XML Bible</title>
      <authors>
         <aut_email>andy@yahoo.com</aut_email>
         <aut_email>ben@yahoo.com</aut_email>
      </authors>
      subject ed_email
   </book>
</xml>
```

g. Transformation of `subject` terminal symbol is as follows:
`subject ed_email` →

```
<xml id="B01">
   <book isbn="9872347765" lang="EN">
      <title>XML Bible</title>
      <authors>
         <aut_email>andy@yahoo.com</aut_email>
         <aut_email>ben@yahoo.com</aut_email>
      </authors>
      <subject>Database</subject>
      ed_email
   </book>
</xml>
```

h. Transformation of `ed_email` terminal symbol is as follows:
`ed_email` →

```
<xml id="B01">
   <book isbn="9872347765" lang="EN">
      <title>XML Bible</title>
      <authors>
         <aut_email>andy@yahoo.com</aut_email>
         <aut_email>ben@yahoo.com</aut_email>
      </authors>
      <subject>Database</subject>
      <ed_email>ross@gmail.com</ed_email>
   </book>
</xml>
```

□

3.3 Operations on Extended Tree Grammars

In a real situation, we may want to perform operations on the structure of a document. Modification of XML structures can be obtained by a merge operation on two ETGs, or a projection operation on an ETG.

3.3.1 Merge of Extended Tree Grammars

An operation to merge two ETGs into an ETG is needed when we intend to combine structures of any two documents into a single structure. The merge operation of two ETGs is defined as follows:

Definition 12. *Let* $G = (N_G, T_G, A_G, S_G, P_G)$ *and* $H = (N_H, T_H, A_H, S_H, P_H)$ *be ETGs,* $Y \subseteq N_G$*,* $Z \subseteq N_H$*, and* $N_G \cap N_H = \emptyset$*. Let* r *be a regular expression over* Y*, and* s *be a regular expression over* Z*. Let* S→xml[id](r) *be a production rule for start symbol in G, and* S→xml[id](s) *be a production rule for start symbol in H. Merge of ETGs is denoted as F=G+H and is an operation that combines G and H, such that* $F = (N, T, A, S, P)$ *is an ETG where* $N = N_G \cup N_H$*,* $T = T_G \cup T_H$*,* $A = A_G \cup A_H$*, and* $P = P_G \cup P_H$*. Production rule for a start symbol in F is* S→xml[id]($r \llcorner s$)*.*

Example 3.12. *A result of merge ETGs in Example 3.7 and 3.8.*

Let G be an ETG for `book` documents (Example 3.7) and H be an ETG for `author` documents (Example 3.8). Merge operation on G and H (G+H) produces the following ETG:

```
N={S,BOOK,TITLE,AUTHORS,AUT_EMAIL,SUBJECT,GENRE,ED_EMAIL,AUTHOR,NAME,EMAIL}
T={xml,book,title,authors,aut_email,subject,genre,ed_email,author,name,email}
A={id,isbn,lang}
P={S→xml[id](BOOK AUTHOR),
   BOOK→book[isbn lang](TITLE AUTHORS (SUBJECT|GENRE) ED_EMAIL?),
   TITLE→title,AUTHORS→authors(AUT_EMAIL+),SUBJECT→subject,GENRE→genre,
   AUT_EMAIL→aut_email,ED_EMAIL→ed_email,
   AUTHOR→author(NAME EMAIL),NAME→name,EMAIL→email}
```

□

The merger of two ETGs from autonomous external sources may face three different issues regarding unification of the symbols. The first issue is when there are duplicate symbols with the same meaning from two ETGs. In order to keep the ETG size compact, we take one of these duplicate symbols, which is quite simple.

Duplicate terminal symbols with the same meaning can be found as a result of a merge operation on two ETGs, Example 3.12 and 3.9. Both ETGs have `name` and `email` elements, and both `name`s and both `email`s have the same meanings. Example 3.13 shows an ETG result after a merge operation on the documents' ETG.

The second issue is when there are duplicate symbols from two ETGs which have different meanings. To solve this problem, we use a naming resolution algorithm in order to keep all symbols and refer to their original meaning when a new ETG is produced as a result. Naming resolution is not covered in this thesis.

The last issue is when there are two different symbols which have the same meaning. In this case, we keep both symbols and treat them as different symbols, for example, the `aut_email` and `email` symbols in Example 3.12.

Furthermore, Figure 3.7 shows an XML document as a result of merging a `book` document in Figure 3.4, an `author` document in Figure 3.5, and an `editor` document in Figure 3.6. In contrast to the terminal symbols, we cannot combine non-terminal symbols. Then, the duplicate non-terminal symbols must be renamed such that they preserve their paths from their original structures.

Corollary 3.1. *Let $G = (N, T, A, S, P)$ be an Extended Tree Grammar, and $N = \{X_1, X_2, ..., X_n\}$. Let $H = (N', T, A, S, P)$ be an Extended Tree Grammar, $N' = (N - \{X_i\}) \cup \{X_i'\}$, and N' is obtained through a systematic renaming $X_i \rightarrow X_i'$. Then the language of G is the same as the language of H, L(G)=L(H).*

Example 3.13. *An ETG result of merger* `book`, `author` *and* `editor` *structures.* Merger of those three ETGs is done by merging `book` and `author` ETGs first as in Example 3.12, and then merging the result with `editor` ETG. Since the ETG in Example 3.12 and the ETG for `editor` in Example 3.9 have the same non-terminal symbol, we apply renaming for non-terminal symbols `NAME` and `EMAIL` in ETG for `editor` data, such that `NAME` becomes `NAME2` and `EMAIL` becomes `EMAIL2`. Renaming of `NAME` → `NAME2` applies renaming of production rule `NAME`→ `name` into `NAME2`→ `name`. Renaming of `EMAIL` → `EMAIL2` applies renaming of production rule `EMAIL`→ `email` into `EMAIL2`→ `email`. After the renaming process, we perform a merge operation on the two ETGs, such that the following ETG comes as a result:

```
N={S,BOOK,TITLE,AUTHORS,AUT_EMAIL,SUBJECT,GENRE,ED_EMAIL,AUTHOR,EDITOR,
   NAME,EMAIL,NAME2,EMAIL2}
T={xml,book,title,authors,aut_email,subject,genre,ed_email,author,editor,
   name, email}
A={id,isbn,lang}
```

```
P={S→xml[id](BOOK AUTHOR EDITOR),
   BOOK→book[isbn lang](TITLE AUTHORS (SUBJECT|GENRE) ED_EMAIL?),
   TITLE→title,AUTHORS→authors(AUT_EMAIL+),
   SUBJECT→subject,GENRE→genre,AUT_EMAIL→aut_email,ED_EMAIL→ed_email,
   AUTHOR→author(NAME EMAIL),EDITOR→editor(NAME2 EMAIL2),
   NAME→name,EMAIL→email,NAME2→name,EMAIL2→email}
```

□

```
<xml id="001">
   <book isbn="9872347765" lang="EN">
      <title>XML</title>
      <authors>
         <aut_email>andy@yahoo.com</aut_email>
         <aut_email>ben@yahoo.com</aut_email>
      </authors>
      <subject>Data</subject>
      <ed_email>ross@gmail.com</ed_email>
   </book>
   <author>
      <name>Andy Cole</name>
      <email>andy@yahoo.com</email>
   </author>
   <editor>
      <name>Ross Marrie</name>
      <email>ross@gmail.com</email>
   </editor>
</xml>
```

Figure 3.7: An example of a more complex XML document

Based on the ETG defined in Example 3.13 we are able to generate the following sentences:

```
xml[id](book[isbn lang](title authors(aut_email aut_email) subject ed_email))
xml[id](book[isbn lang](title authors(aut_email) subject))
xml[id](author(name email))
xml[id](editor(name email))
```

Merge of two ETGs is required to perform *join* operation later on.

3.3.2 The Projection of an Extended Tree Grammar

The projection of an ETG is used to modify the structure of an XML document by removal of some elements from its original structure. Using an example of

XML document in Figure 3.7, a number of removal operations can be applied to manipulate its structure, such as:

1. Removing a particular element, for example removal of all `email` elements.
2. Removing an element in a specific path, for example removal of the `email` element under the `author` subtree.
3. Removing a subtree, for example removal of a subtree rooted at the `author` element.
4. Retrieving a subtree, for example retrieval of a subtree rooted at the `author` element.

Definition 13. *Let $G = (N, T, A, S, P)$ be an ETG, and $N = \{X_1, \ldots, X_n, Y_1, \ldots, Y_n\}$ be a set of non-terminal symbols. Let $Z \subset N = \{Y_1, \ldots, Y_n\}$, $X_i \neq S$, $\overline{X_i} = N - \{X_i\}$ and $A_1, \ldots, A_n \subseteq A$. Let $r(\ldots, X_i, \ldots)$ be a regular expression over a non-terminal symbol X_i and other non-terminal symbols in N. Let G has of the following production rules:*
$X_1 \rightarrow t_1[A_1](r(\ldots, X_i, \ldots))$,
$X_i \rightarrow t_i[A_i](s_1(\mathrm{Z}))$,
$X_i \rightarrow t_i[A_j](s_2(\mathrm{Z}))$,
$\ldots$,
$X_i \rightarrow t_i[A_m](s_n(\mathrm{Z}))$
where $s_i(Z)$ is a regular expression over Z. Projection of G on $N - \{X_i\}$, denoted as $\pi_{\overline{X_i}}(G)$, and is defined as modification of G through removal of a non-terminal symbol X_i at right hand side of a production rule $X_1 \rightarrow t_1[A_1](r(\ldots, X_i, \ldots))$ such that it becomes:
$X_1 \rightarrow t_1[A_1](r(\ldots, s_1(\mathrm{Z}), \ldots))$,
$X_1 \rightarrow t_1[A_1](r(\ldots, s_2(\mathrm{Z}), \ldots))$,
$\ldots$,
$X_1 \rightarrow t_1[A_1](r(\ldots, s_n(\mathrm{Z}), \ldots))$.

Example 3.14. *Projection of an Extended Tree Grammar.*

The result of a projection operation to remove a non-terminal symbol `AUTHORS` from an ETG in Example 3.13 is as follows:

```
N={S,BOOK,TITLE,AUT_EMAIL,SUBJECT,GENRE,ED_EMAIL,AUTHOR,EDITOR,NAME,EMAIL,
   NAME2,EMAIL2}
T={xml,book,title,aut_email,subject,genre,ed_email,author,editor,name,email}
A={id,isbn,lang}
```

```
P={S→xml[id](BOOK AUTHOR EDITOR),
   BOOK→book[isbn lang](TITLE AUT_EMAIL+ (SUBJECT|GENRE) ED_EMAIL?),
   TITLE→title,AUT_EMAIL→aut_email,SUBJECT→subject,GENRE→genre,
   ED_EMAIL→ed_email,AUTHOR→author(NAME EMAIL),EDITOR→editor(NAME2 EMAIL2),
   NAME→name,EMAIL→email,NAME2→name,EMAIL2→email}
```

□

The document in Figure 3.7 can be transformed into new documents which are subsets of the original document. Figure 3.8 illustrates transformation rules of an XML document tree.

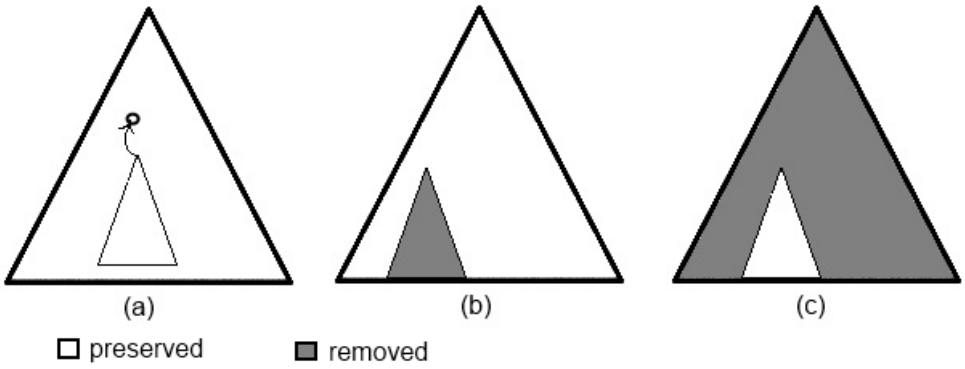

Figure 3.8: Transformation rules (a) removal of a level (b) removal of a sub-tree (c) extraction of a sub-tree

Definition 14. *Let* $H = (N_H, T_H, A_H, S_H, P_H)$ *and* $G = (N_G, T_G, A_G, S_G, P_G)$ *be ETGs. Let* X *be a set of non-terminal symbols,* $X \subseteq N_G$*, and* $\neg\exists Y \in X : Y \in N_H$*.* H *is a sub-grammar of* G*, denoted as* $H \sqsubseteq G$ *if* $N_H \subseteq N_G, T_H \subseteq T_G, A_H \subseteq A_G$ *and* H *can be obtained from* G *by multiple applications of projection on* $\overline{X}$*.*

Example 3.15. *Transformation of an ETG to remove a single element*
Transformation of an ETG to remove the `authors` element in Example 3.13 is performed by multiple applications of projection over the ETG, and can be achieved using the following steps:

1. Find all production rules which contain a non-terminal symbol `AUTHORS` at its right hand side. It returns a production rule:
   `BOOK→book(TITLE AUTHORS (SUBJECT|GENRE) ED_EMAIL?)`

2. Find a production rule for a non-terminal symbol `AUTHORS`.
   `AUTHORS→author(AUT_EMAIL+)`

3. Obtain a non-terminal symbol string from the right hand side of the production rule (AUT_EMAIL+).

4. Replace the non-terminal symbol AUTHORS from the production rule BOOK→book(TITLE AUTHORS (SUBJECT|GENRE) ED_EMAIL), such that it becomes
   BOOK→book(TITLE AUT_EMAIL+ (SUBJECT|GENRE) ED_EMAIL?)

A projection operation on ETG to remove a non-terminal symbol AUTHORS results in the following ETG:

N={S,BOOK,TITLE,AUT_EMAIL,SUBJECT,GENRE,ED_EMAIL,AUTHOR,EDITOR,NAME,EMAIL,
NAME2,EMAIL2}
T={xml,book,title,aut_email,subject,genre,ed_email,author,editor,name,email}
A={id,isbn,lang}
P={S→xml[id](BOOK),
BOOK→book[isbn lang](TITLE AUT_EMAIL+ (SUBJECT|GENRE) ED_EMAIL),
TITLE→title,AUT_EMAIL→aut_email,
SUBJECT→subject,GENRE→genre,ED_EMAIL→ed_email,
AUTHOR→author(NAME EMAIL),EDITOR→editor(NAME EMAIL),
NAME→name,EMAIL→email,NAME2→name,EMAIL2→email}

□

Example 3.16. *Removal of a sub-tree rooted a particular non-terminal symbol.* Transformation of an ETG to remove a subtree rooted at AUTHOR under S is performed by the following steps:

1. Find a production rule for a non-terminal symbol S:
   S→xml[id](BOOK AUTHOR EDITOR).

2. Remove non-terminal symbol AUTHOR on the right hand side of the production rule such that it becomes S→xml[id](BOOK EDITOR).

3. Application of multiple removals of a non-terminal symbol on the right hand site of the production rule of AUTHOR allows us to delete non-terminal symbols as well as production rules in the subtree if needed.

After removal of a subtree rooted at element authors, the ETG of the BOOK document becomes:

N={S,BOOK,TITLE,AUTHORS,AUT_EMAIL,SUBJECT,GENRE,ED_EMAIL,EDITOR,NAME2,EMAIL2}
T={xml,book,title,authors,aut_email,subject,genre,ed_email,editor,name,email}

```
A={id,lang}
P={S→xml[id](BOOK EDITOR),
   BOOK→book[isbn lang](TITLE AUTHORS (SUBJECT|GENRE) ED_EMAIL?),
   TITLE→title,AUTHORS→authors(AUT_EMAIL+),
   SUBJECT→subject,GENRE→genre,AUT_EMAIL→aut_email,ED_EMAIL→ed_email,
   EDITOR→editor(NAME2 EMAIL2),NAME2→name,EMAIL2→email}
```

□

Example 3.17. *Extraction of a sub-tree rooted at a non-terminal symbol.*
Extraction of a subtree rooted at `AUTHOR` which is below a start symbol `S` can be performed in two ways:

1. By modification of a production rule of a non-terminal symbol `S` into `S→xml[id](AUTHOR)`. It is faster when the size of the extracted sub-tree is much smaller than the entire document, but in this way we cannot remove non-terminal symbols and production rules which are not needed.

2. By multiple removals of a level from the top most level until only the extracted sub-tree remains. It is faster when the extracted sub-tree covers most parts of the original document. Extraction in this way will remove unassigned non-terminal symbols and production rules.

After extraction of a sub-tree rooted at `AUTHOR`, the ETG becomes as follows:

```
N={S,AUTHOR,NAME,EMAIL}
T={xml,author,name,email}
A={id}
P={S→xml[id](AUTHOR),
   AUTHOR→author(NAME EMAIL),
   NAME→name,EMAIL→email}
```

□

The ETG in Example 3.15-3.17 are sub-grammars of the ETG in Example 3.13, because they can be obtained through application of one of the transformation rules.

3.4 XML Algebra for Online Processing

The semistructured data integration system requires elementary operations on the containers with semistructured data. A set of algebraic operations on the XML documents presented in this section allows for incremental processing of semistructured data against the entire XML document.

3.4.1 Data Container and Schema

We use the concept of a data container which is a collection of XML documents, to enable processing of XML documents. In the relational model, a simple data container can be conceptually implemented as a relational table. It has a schema, which requires to all data in the table to follow the schema. In a semistructured data environment we have more freedom to define the structure of the document rather than in a relational model. The concept of schema gives us the opportunity to have instance documents with various structures as long as they are consistent with the defined schema.

Definition 15. *A single schema data container* $D(\{G\})$ *is defined as* $\{x(m) : \exists m\ m \in L(G)\}$ *where* $G = (N, T, A, S, P)$ *be an ETG,* m *be a sentence in* $L(G)$*, and* $x(m)$ *be an XML document.*

In a distributed multi-database system, having a global schema for all external sites is not feasible as remote sites are autonomous. Then, it may happen that we required to combine two data containers with different schemas. To be consistent with the definition of a single schema data container, we have to generate a single result schema such that it covers both schemas from all the data containers. It can be obtained by merging two ETG into a new ETG. Example 3.18 shows how a new ETG can be produced when two single schema data containers are combined.

Example 3.18. *A merged Extended Tree Grammars.*
For example, we combine two data containers where each has a single schema. The first data container requires an author data to have a `name` and at least one `email` data. Meanwhile, the other data container requires an author to have `FirstName`, `LastName` and at least one `email` data. The ETG of the single schema data containers can be shown as follows:

```
N={S,AUTHOR,NAME,EMAIL}
T={xml,author,name,email}
P={S→xml(AUTHOR),AUTHOR→author(NAME,EMAIL+),
   NAME→name,EMAIL→email}

N={S,AUTHOR,FIRSTNAME,LASTNAME,EMAIL}
T={xml,author,firstname,lastname,email}
P={S→xml(AUTHOR),AUTHOR→author(FIRSTNAME,LASTNAME,EMAIL+),
   FIRSTNAME→firstname,LASTNAME→lastname,EMAIL→email}
```

Then, result of merging ETGs is as follows:

N={S,AUTHOR,NAME,FIRSTNAME,LASTNAME,EMAIL}
T={xml,author,name,firstname,lastname,email}
P={S→xml(AUTHOR),AUTHOR→author((NAME|(FIRSTNAME,LASTNAME)),EMAIL+),
NAME→name,FIRSTNAME→firstname,LASTNAME→lastname,EMAIL→email}

□

In general, we are able to combine two single data containers even if they have totally different schemas. Some potential problems may arise when we combine two ETGs into a ETG, such as:

1. Two non-terminal symbols represent the same meaning, for example although NAME and FULLNAME are different non-terminal symbols, they may have the same meaning. Then, we must have a naming definition and mapping such that there is enough information on which symbols have similar meaning.

2. A non-terminal symbol represents a different meaning in each ETG. For example, the non-terminal symbol TITLE in book document has different meaning to title in author data. To overcome this problem, we need to rename the non-terminal symbols or apply indexing to the terminal symbols.

3. A non-terminal symbol has more than one production rule, and each of them points to a different terminal symbol (for example: X→ t_1(r(N))) and X→ t_2(r(N)))). Non-terminal symbol indexing or renaming can be a feasible solution to this issue.

Mapping and naming resolution for non-terminal symbols can be a painful solution when dealing with complex operations. Therefore, it may not be a good decision to force the unification of ETGs when combining documents from data containers.

Another possible solution is to collect all ETGs which come along with XML documents from both data containers. This means that a data container result may have a set of schemas rather than a single schema, which can avoid the problem with symbol mapping and naming resolution described earlier. Then, to be consistent, a data container is redefined in Definition 16.

Definition 16. *Let $\mathcal{G}$ be a set of ETGs. A data container $D(\mathcal{G})$ is defined as $\{x(m) : \exists G \exists m\ G \in \mathcal{G} \wedge m \in L(G)\}$ where G=(N,T,A,S,P) be a ETG and $x(m)$ be an XML document. $\mathcal{G}$ is a schema of a data container.*

Example 3.19. *A data container with a set of ETGs as its schema.*
For example, we place two XML documents in a data container $D(\mathcal{G})$. An XML document is consistent with an ETG G as follows:

```
N={S,AUTHOR,NAME,EMAIL}
T={xml,author,name,email}
P={S→xml(AUTHOR),AUTHOR→author(NAME,EMAIL+),
   NAME→name,EMAIL→email}
```

And the other XML document is consistent with an ETG H as follows:

```
N={S,AUTHOR,FIRSTNAME,LASTNAME,EMAIL}
T={xml,author,firstname,lastname,email}
P={S→xml(AUTHOR),AUTHOR→author(FIRSTNAME,LASTNAME,EMAIL+),
   FIRSTNAME→firstname,LASTNAME→lastname,EMAIL→email}
```

Data container $D(\mathcal{G})$ has a schema $\mathcal{G}$ such that $\mathcal{G} = \{G\} + \{H\}$. □

The schema of a data container plays an important role when computation of operations are being executed. The schema allows us to verify whether a particular structure can be found in a set of documents without checking every instance of the document, and gives a better performance in the computation of large size data containers.

3.4.2 XML Algebra Operations

The operators of XML algebra operate on the data containers, ETGs, paths, and a number of condition functions. All operations return a data container which automatically generates a new identifier for every XML document result. The system includes a set of basic operators: {*projection* (π), *selection* (σ), *join* ($\bowtie$), *antijoin* ($\sim$), and *union* ($\cup$)}. The operators are conceptually consistent with the basic operators of relational algebra.

3.4.2.1 Projection operation

The projection operation on a data container can be described from a projection operation on an XML document, and can then be expanded. Definition 19 defines the projection operation on a single XML document. Whereas, Example 3.23 represents a projection on a single XML document.

Definition 17. *Let $G = (N, T, A, S, P)$ be an ETG, $m \in (T \cup A)*$ be a sentence of G. $m' \in (T' \cup A')*$ is a sub-sentence of m, denoted as $m' \sqsubseteq m$, and is defined as result of removal $t \in T$ and $a \in A$ from m such that $T' \subset T$ and $A' \subseteq A$.*

Example 3.20. *Sub-sentences of a sentence* m=`xml[id](book[isbn lang](title authors(aut_email aut_email) subject ed_email))`

Let G be an ETG, and m be a sentence of G. The first sub-sentence example can be obtained by the removal of `ed_email` terminal symbol from m such that:
m' =`xml[id](book[isbn lang](title authors(aut_email aut_email) subject))`
The next sub-sentence example is by removal of the `book` terminal symbol from m, which will automatically remove its attributes. After the removal of `book` element we obtain:
m' =`xml[id]((title authors(aut_email aut_email) subject ed_email))`
Brackets which follow `book` element can be eliminated such that it becomes:
m' =`xml[id](title authors(aut_email aut_email) subject ed_email)` □

In the case of a sub-sentence m' =`xml[id](book [isbn lang](title authors (aut_email aut_email) subject))` in Example 3.20, both m and m' are sentences of G. But, it is important to notice that even though $m' \sqsubseteq m$, this does not mean that m' is always a sentence of G. The second sub-sentence example m' =`xml[id](title authors(aut_email aut_email) subject ed_email)` does not satisfty G, therefore it is not a sentence of G.

On the other hand, when an ETG H is a sub-grammar of an ETG G, a sentence $n \in$H is likely be a sub-sentence of any sentence $m \in$G. But, having H$\sqsubseteq$G, we cannot conclude that $n \sqsubseteq m$.

Example 3.21. *Relation between sub-sentence and sub-grammar*

Let G be an ETG as in Figure 3.7, H be an ETG and $H = G$. m be a sentence of G, n be a sentence of H.
Let m=`xml[id](book[isbn lang](title authors(aut_email) subject))`.
Let n=`xml[id](book[isbn lang](title authors(aut_email) subject ed_email))`.
Although H is a sub-grammar of G (H$\sqsubseteq$G), n is not a sub-sentence of m ($n \not\sqsubseteq m$). □

Definition 18. *Let $G = (N, T, A, S, P)$ be an ETG, $Y \in N$ be a non terminal symbol, m be a sentence in $L(G)$, and $x(m)$ be an XML document. Let $\overline{Y} = N - \{Y\}$. Projection of $x(m)$ on $\overline{Y}$ is denoted as $x(m)[\overline{Y}] = x(m')$ where $G' = (N', T', A', S', P')$, $G' \sqsubseteq G$, $m' = m[\overline{Y}]$, $m' \sqsubseteq m$, m' is a sentence in $L(G')$, and $Y \notin N'$. $m[\overline{Y}]$ is performed by removal of terminal symbol which corresponds to projection of G on $\overline{Y}$. To remove a single element in an XML document, projection operation is perfomed in two steps:*

1. *Transformation of a sentence m into m'*

2. *Transformation m' into an XML document $x(m')$ in the same sequence of transformation m into $x(m)$.*

A new **`id`** *attribute value is assigned to $x(m')$ to represent a unique immutable identity.*

Example 3.22. *Projection (π) operation on an XML document to remove one element*
Let $x(m)$ be an XML document as shown in Figure 3.4, where m=`xml[id](book[isbn lang](title authors(aut_email aut_email) subject ed_email))`. Let G be an ETG as shown in Example 3.7, Y∈N be non-terminal symbol `AUTHORS`. Projection of $x(m)$ on $\overline{Y}$ is performed as follows:

1. First, projection of G on $\overline{Y}$ results on removal a production rule: `AUTHORS`→ `authors(AUT_EMAIL+)` and replacement of a production rule: `BOOK`→ `book(TITLE AUTHORS (SUBJECT|GENRE) ED_EMAIL?)` with: `BOOK`→ `book(TITLE AUT_EMAIL+ (SUBJECT|GENRE) ED_EMAIL?)`. Transformation of a sentence m is performed by removal of a terminal symbol `authors` in `book`. Then, m'=`xml[id](book[isbn lang](title aut_email aut_email subject ed_email))`.

2. The next step is to transform m' into an instance of the XML document. It is performed step by step as in Example 3.11 except step (d) as follows:

   (a) First, transformation of the `xml[id]` terminal symbol from a sentence: `xml[id](book[isbn lang](title aut_email aut_email subject ed_email))`→

   ```
   <xml id="A01">
      book[isbn lang](title authors(aut_email aut_email) subject
      ed_email)
   </xml>
   ```

   (b) Next, transformation of the `book[isbn lang]` terminal symbol as follows: `book[isbn lang](title aut_email aut_email subject ed_email)`→

   ```
   <xml id="A01">
      <book isbn="9872347765" lang="EN">
         title authors(aut_email aut_email) subject ed_email
      </book>
   </xml>
   ```

   (c) Then, transformation of the `title` terminal symbol as follows: `title aut_email aut_email subject ed_email` →

```
<xml id="A01">
   <book isbn="9872347765" lang="EN">
      <title>XML Bible</title>
      aut_email aut_email subject ed_email
   </book>
</xml>
```

(d) Transformation of the first `aut_email` terminal symbol from:
`aut_email aut_email subject ed_email` →

```
<xml id="A01">
   <book isbn="9872347765" lang="EN">
      <title>XML Bible</title>
      <aut_email>andy@yahoo.com</aut_email>
      aut_email subject ed_email
   </book>
</xml>
```

(e) Transformation of the second `aut_email` terminal symbol as follows:
`aut_email subject ed_email` →

```
<xml id="A01">
   <book isbn="9872347765" lang="EN">
      <title>XML Bible</title>
      <aut_email>andy@yahoo.com</aut_email>
      <aut_email>ben@yahoo.com</aut_email>
      subject ed_email
   </book>
</xml>
```

(f) Transformation of the `subject` terminal symbol as follows:
`subject ed_email` →

```
<xml id="A01">
   <book isbn="9872347765" lang="EN">
      <title>XML Bible</title>
      <aut_email>andy@yahoo.com</aut_email>
      <aut_email>ben@yahoo.com</aut_email>
      <subject>Database</subject>
      ed_email
   </book>
</xml>
```

(g) Transformation of the `ed_email` terminal symbol as follows:
`ed_email` →

```
<xml id="A01">
   <book isbn="9872347765" lang="EN">
      <title>XML Bible</title>
      <aut_email>andy@yahoo.com</aut_email>
      <aut_email>ben@yahoo.com</aut_email>
      <subject>Database</subject>
```

```
        <ed_email>ross@gmail.com</ed_email>
    </book>
</xml>
```

□

Definition 19. *Let $G = (N, T, A, S, P)$, $H = (N', T', A', S', P')$ be ETGs, and $H \sqsubseteq G$. Let m be a sentence in $L(G)$, and $x(m)$ be an XML document with a structure defined by G. Let $M = N - N' = \{M_1, M_2, \ldots, M_n\}$ be a set of non-terminal symbols removed from G. Projection of x(m) on a sub-grammar H is a unary operator denoted by $x(m)[H] = ((x(m)[\overline{M_1}])[\overline{M_2}])...[\overline{M_n}]$, Where $x(m)[\overline{M_i}]$ is projection of $x(m)$ on $\overline{M_i}$ where $i = 1, \ldots, n$. A new* **id** *attribute value is assigned to $x(m')$ to represent a unique immutable identity.*

Example 3.23. *Projection (π) operation of an XML document on an ETG*
Let $x(m)$ be an XML document as shown in Figure 3.4 with an ETG G as shown in Example 3.7. Let H⊑G be an ETG as follows:

```
N={S,BOOK,TITLE,SUBJECT}
T={xml,book,title,subject}
A={id,isbn,lang}
P={S→xml[id](BOOK),BOOK→book[isbn lang](TITLE SUBJECT),
   TITLE→title,SUBJECT→subject}
```

The projection of XML document $x(m)$ on H is performed by the following steps:

1. First we check whether H⊑G. If H⊑G then we apply ETG projection on H to the XML document $x(m)$. Otherwise, no document is returned as a result.

2. Obtain a set of non-terminal symbols which are removed from G. $M = N - N' =$ {AUT_EMAIL,AUTHORS,GENRE,ED_EMAIL}

3. Perform the projection of $x(m)$ on every non-terminal symbol in M. After projection on a single element, we get an XML document with a new ETG. Then we perform the next non-terminal symbol based on the new ETG. Another option is to remove all non-terminal symbols without creating an XML document for every single removal. This can be performed by removal from the deepest level, therefore no ETG modification is required for every symbol removal.

4. We may notice that the non-terminal symbol `SUBJECT` is an alternative to `GENRE`. If the sentence m contains `genre` which is a derivation of `GENRE`, then the document does not satisfy H, and therefore should not be taken as a result.

5. With the structure of document $x(m)$: m=`xml[id](book[isbn lang](title authors(aut_email aut_email) subject ed_email))`, we apply the following steps:

   a. First, we remove the `aut_email` symbol in the `authors` element

   b. Then, we remove the `authors` symbol in the `book` element

   c. Last, we remove the `ed_email` symbol in the `book` element

After the projection operation, the structure of the result document is defined by a sentence m' =`xml[id](book[isbn lang](title subject))`. Then, based on the new sentence m', we instantiate an XML document by transformation of m' as in Example 3.11 except for steps (d), (e), (f) and (h) as follows:

1. First, transformation of the `xml[id]` terminal symbol from a sentence: `xml[id](book [isbn lang](title subject))`→

```
<xml id="A01">
   book[isbn lang](title subject)
</xml>
```

2. Next, we transform the `book[isbn lang]` terminal symbol such that: `book[isbn lang](title subject)`→

```
<xml id="A01">
   <book isbn="9872347765" lang="EN">
      title authors(aut_email aut_email) subject ed_email
   </book>
</xml>
```

3. Then, transformation of the `title` terminal symbol from:
`title subject` →

```
<xml id="A01">
   <book isbn="9872347765" lang="EN">
      <title>XML Bible</title>
      subject
   </book>
</xml>
```

4. Transformation of the `subject` terminal symbol from:
`subject ed_email` →

```
<xml id="A01">
   <book isbn="9872347765" lang="EN">
      <title>XML Bible</title>
      <subject>Database</subject>
   </book>
</xml>
```

□

Since a data container is a set of XML documents, then projection on a data container can be performed by iterative projections on each data container member. Projection operation on a data container is defined as in Definition 20.

Definition 20. *Let $D(\mathcal{G})$ be a data container of XML documents, and $x_i(m_i) \in D(\mathcal{G})$. Projection on $D(\mathcal{G})$ is a unary operator denoted by $\pi_H(D(\mathcal{G})) = \{y(m') : \exists x(m) \in D(\mathcal{G}), y(m') = x(m)[H]\}$.*

Example 3.24. *Projection (π) operation over a data container*
Let $D(\mathcal{G})$ be a data container where $\mathcal{G}$ is a single set of ETG, as in Example 3.13. $D(\mathcal{G})$ contains a single XML document `BOOK` as in Figure 3.7. Let H be an ETG as in Example 3.17 and $H \in \mathcal{H}$. The projection operation $\pi_H(D(\mathcal{G}))$ produces a result of a data container $D_r(\mathcal{H})$. Since $D(\mathcal{G})$ contains a single document, then the result data container has at most one XML document. The new XML document $x(m')$ is assigned with a new `id` attribute as a unique immutable identity as indicated in Figure 3.9. □

```
<xml id="R01">
   <author>
      <name>Andy Cole</name>
      <email>andy@yahoo.com</email>
   </author>
</xml>
```

Figure 3.9: Result of the projection operation (π) on a data container $D(\mathcal{G})$

The projection on a single XML document ($\pi_H(x(m))$) has an operation to check the inclusion of H in an ETG of $x(m)$. Since a data container $D(\mathcal{G})$ has a set of ETGs as its schema, we can increase the performance of the projection operation by first finding which ETG has a sub-grammar match with an ETG H. Some documents can be pre-eliminated.

3.4.2.2 Selection operation

Since a data container is defined as a set of XML documents, then selection on a data container can be performed by iterative selections on each data container member. The selection operation on a data container is defined in Definition 22.

Definition 21. *Let p be a path expression, and v be a string value. Let θ be an operator in a set of comparison operator $\{=, <>, <, >, <=, >=\}$. A condition expression is denoted as φ, and is defined as a propositional formula that consists of proposition(s) in the form of $(p\ \theta\ p)$ or $(p\ \theta\ v)$, and the logical operators $\wedge$ (and), $\vee$ (or) and $\neg$ (negation).*

Definition 22. *Let $D(\mathcal{G})$ be a data container of XML documents, and $x(m) \in D(\mathcal{G})$. Selection on $D(\mathcal{G})$ is a unary operator denoted by $\sigma_{\varphi}(D(\mathcal{G})) = \{x(m) : f(x(m), \varphi) = true\}$, where $f(x(m), \varphi)) \in \{true, false\}$ and φ is a condition expression. The value of attribute id in resulting documents $x(m)$ is replaced with new values. The result is a data container with schema $\mathcal{H} \subseteq \mathcal{G}$.*

Example 3.25. *Selection (σ) operation on a data container*
Let $D(\mathcal{G})$ be a data container where $\mathcal{G}$ is a single set of ETG, as in Example 3.13. $D(\mathcal{G})$ contains a single XML document `BOOK` as in Figure 3.7. Let φ be a condition expression `/author/name="Andy Cole"`. The selection operation $\sigma_{\varphi}(D(\mathcal{G}))$ produces a data container $D(\mathcal{H}) = D(\mathcal{G})$ since $D(\mathcal{G})$ only contains one document and it satisfies the condition (φ). The resulting XML document is assigned a new `id` attribute as a unique immutable identity, as shown in Figure 3.10. □

Binary operations deal with two database containers as their arguments. In the definitions below, $D(\mathcal{G})$ and $D(\mathcal{H})$ denote data containers of XML documents, $G = (N_G, T_G, A_G, S_G, P_G) \in \mathcal{G}$ and $H = (N_H, T_H, A_H, S_H, P_H) \in \mathcal{H}$. $D(\mathcal{F})$ is a resulting data container.

To provide a better understanding, we assume a data container $D(\mathcal{G})$ contains two `BOOK` XML documents as in Figure 3.4, and data container $D(\mathcal{H})$ contains three `AUTHOR` XML documents, as in Figure 3.5.

3.4.2.3 Union operation

In the relational model, the *union* operation operates on two relational tables which have the same number of attributes and data types (schema). In this work, the *union* operator allows an operation on data containers with different schemas, and the data container result has a collection of ETGs from both input data containers as its schema.

```
<xml id="S01">
   <book isbn="9872347765" lang="EN">
      <title>XML</title>
          <authors>
         <aut_email>andy@yahoo.com</aut_email>
         <aut_email>ben@yahoo.com</aut_email>
     </authors>
      <subject>Data</subject>
      <ed_email>ross@gmail.com</ed_email>
   </book>
   <author>
      <name>Andy Cole</name>
      <email>andy@yahoo.com</email>
   </author>
   <editor>
      <name>Ross Marrie</name>
      <email>ross@gmail.com</email>
   </editor>
</xml>
```

Figure 3.10: Result of the selection operation (σ) on a data container $D(\mathcal{G})$

The *union* operation over data containers is defined in Definition 23.

Definition 23. *Let $D(\mathcal{G}), D(\mathcal{H})$ be data containers. Union operator is defined as $D(\mathcal{G}) \cup D(\mathcal{H}) = \{x(m) : x(m) \in D(\mathcal{G})$ or $x(m) \in D(\mathcal{H})\}$. The resulting data container has a schema $\mathcal{F} = \mathcal{G} \cup \mathcal{H}$. XML document $x(m) \in D(\mathcal{F})$ will be assigned to a new `id` attribute value which represents a unique identity.*

A *union* operation takes all documents from both data containers and places the documents in a data container as a result. After a *union* operation, the resulting data container has a schema which is the unification of both data containers' schemas.

Example 3.26. *Union ($\cup$) operation over two data containers*
The *union* operation ($D(\mathcal{G}) \cup D(\mathcal{H})$) results in a data container with a schema $\mathcal{F} = \mathcal{G} \cup \mathcal{H}$, and XML documents as in Figure 3.11. □

3.4.2.4 Join operation

The next binary operation is the *join* operation. Firstly, we describe a *join* operation on two XML documents, then we expand it to operate on two data containers. The *join* operation on two XML documents is defined in Definition 24.

```
<xml id="U01">
   <book isbn="9872347765" lang="EN">
      <title>XML</title>
      <authors>
         <aut_email>andy@yahoo.com</aut_email>
         <aut_email>ben@yahoo.com</aut_email>
      </authors>
      <subject>Data</subject>
      <ed_email>ross@gmail.com</ed_email>
   </book>
</xml>

<xml id="U02">
   <book isbn="9788700631625" lang="EN">
      <title>Harry Potter and the Philosopher's Stone</title>
      <authors>
         <aut_email>jk@yahoo.com</aut_email>
      </authors>
      <genre>Fantasy</genre>
   </book>
</xml>

<xml id="U03">
   <author>
      <name>Andy Cole</name>
      <email>andy@yahoo.com</email>
   </author>
</xml>

<xml id="U04">
   <author>
      <name>Ben Johnson</name>
      <email>ben@yahoo.com</email>
   </author>
<xml>

<xml id="U05">
   <author>
      <name>J.K. Rowling</name>
      <email>jk@yahoo.com</email>
   </author>
</xml>
```

Figure 3.11: XML documents in a resulting data container of union operation ($\cup$)

Definition 24. *Let $x(m)$ be an XML document which is consistent with an ETG G, and $y(n)$ be an XML document which is consistent with an ETG H. Let m=`xml[id]`(m') and n=`xml[id]`(n'). The join operation on two XML documents is defined as $x(m) \bullet_{\varphi} y(n) = z(o) : o =$ `xml[id]`$(m' \lrcorner n')$ and $f((x(m), y(n)), \varphi) = true$, where φ is a condition expression and f is an evaluation function such that $f((x(m), y(n)), \varphi) \in \{true, false\}$. An XML document $z(o)$ is obtained through transformation of the sentence o which refers to transformation of m and n. $z(o)$ is assigned to a new `id` attribute value to represent a unique immutable identity. If $f((x(m), y(n)), \varphi)$ is false then the join operation $(\bullet)$ cannot be computed.*

Example 3.27. *Join* $(\bullet)$ *operation over two XML documents*
Let $x(m)$ be an XML document as in Figure 3.4 (with `id`="B01") and $y(n)$ be an XML document as in the first document of Figure 3.5 (with `id`="A01"). Let φ be a condition expression `book//aut_email=author/email`. The *join* operation $x(m) \bullet_{\varphi} y(n)$ results in an XML document as shown in Figure 3.12. □

```
<xml id="J01">
   <book isbn="9872347765" lang="EN">
      <title>XML</title>
      <authors>
         <aut_email>andy@yahoo.com</aut_email>
         <aut_email>ben@yahoo.com</aut_email>
      </authors>
      <subject>Data</subject>
      <ed_email>ross@gmail.com</ed_email>
   </book>
   <author>
      <name>Andy Cole</name>
      <email>andy@yahoo.com</email>
   </author>
<xml>
```

Figure 3.12: An XML document as the result of a *join* operation over two XML documents

A *join* operation over two XML documents can be performed as follows:

1. Get two XML documents $x(m)$ and $y(n)$.
2. Verify whether a condition expression φ is satisfied for both XML documents.
3. If yes, then apply a merger of ETGs to $x(m)$ and $y(n)$, then return the resulting document with a new unique `id` attribute value.

4. Otherwise, return nothing.

Next, we expand the *join* operation over two data containers as in Definition 25.

Definition 25. *Let $D(\mathcal{G}), D(\mathcal{H})$ be data containers. The join operation is defined as $D(\mathcal{G}) \bowtie_{\varphi} D(\mathcal{H}) = \{z(o) : \exists x(m) \in D(\mathcal{G}), y(n) \in D(\mathcal{H}), z(o) = x(m) \bullet_{\varphi} y(n)\}$. The schema of result data container is $\mathcal{F} = \{G + H : \exists x(m) \in D(\mathcal{G}), y(n) \in D(\mathcal{H})$,G is ETG of $x(m)$,H is ETG of $y(n)$, and $x(m) \bullet_{\varphi} y(n)$ satisfies condition expression$\}$.*

Example 3.28. *The join ($\bowtie$) operation over two data containers*
A *join* operation $D(\mathcal{G}) \bowtie_{\varphi} D(\mathcal{H})$, where $x(m) \in D(\mathcal{G})$, $y(n) \in D(\mathcal{H})$, and $\varphi = x(m)$`//aut_email=`$y(n)$`//email` results in a data container $D(\mathcal{F})$, where $\mathcal{F}$ is a single set ETG {F=(N,T,A,S,P)} as follows:

```
N={S,BOOK,TITLE,AUTHORS,AUT_EMAIL,SUBJECT,GENRE,ED_EMAIL,AUTHOR,NAME,EMAIL}
T={xml,book,title,authors,aut_email,subject,genre,ed_email,author,name,email}
A={id,isbn,lang}
P={S→xml[id](BOOK AUTHOR),
   BOOK→book[isbn lang](TITLE AUTHORS (SUBJECT|GENRE) ED_EMAIL?),
   TITLE→title,AUTHORS→authors(AUT_EMAIL+),SUBJECT→subject,GENRE→genre,
   AUT_EMAIL→aut_email,ED_EMAIL→ed_email,AUTHOR→author(NAME EMAIL),
   NAME→name,EMAIL→email}
```

Then, the operation produces a data container which consists of three XML documents, as shown in Figure 3.13 □

3.4.2.5 Antijoin operation

The *antijoin* operation over data containers is defined in Definition 26.

Definition 26. *Let $D(\mathcal{G}), D(\mathcal{H})$ be data containers, $x(m)$ and $y(n)$ be XML documents. The Antijoin operator is defined as $D(\mathcal{G}) \sim_{\varphi} D(\mathcal{H}) = \{x(m) : x(m) \in D(\mathcal{G})$ and $\forall y(n) \in D(\mathcal{H}) \neg\exists (x(m) \bullet_{\varphi} y(n))\}$, where φ is a condition expression. The resulting data container has a schema $\mathcal{F} \subseteq \mathcal{G}$. A new* `id` *attribute value is assigned to every XML document result to represent a unique immutable identity.*

Example 3.29. *Antijoin ($\sim$) operation on data containers*
The *antijoin* operation $D(\mathcal{G}) \sim_{\varphi} D(\mathcal{H})$ where $x(m) \in D(\mathcal{G})$, $y(n) \in D(\mathcal{H})$, and $\varphi = x(m)$`//aut_email=`$y(n)$`//email` results an empty data container $D(\mathcal{F})$. Then the schema of the result data container is $\mathcal{F} = \emptyset$. □

```
<xml id="J01">
   <book>
      <title lang="EN">XML</title>
      <authors>
         <aut_email>andy@yahoo.com</aut_email>
         <aut_email>ben@yahoo.com</aut_email>
      </authors>
      <subject>Data</subject>
      <ed_email>ross@gmail.com</ed_email>
   </book>
   <author>
      <name>Andy Cole</name>
      <email>andy@yahoo.com</email>
   </author>
</xml>
<xml id="J02"
   <book>
      <title lang="EN">XML</title>
      <authors>
         <aut_email>andy@yahoo.com</aut_email>
         <aut_email>ben@yahoo.com</aut_email>
      </authors>
      <subject>Data</subject>
      <ed_email>ross@gmail.com</ed_email>
   </book>
   <author>
      <name>Ben Johnson</name>
      <email>ben@yahoo.com</email>
   </author>
</xml>
<xml id="J03">
   <book isbn="9788700631625" lang="EN">
      <title>Harry Potter and the Philosopher's Stone</title>
      <authors>
         <aut_email>jk@yahoo.com</aut_email>
      </authors>
      <genre>Fantasy</genre>
   </book>
   <author>
      <name>J.K. Rowling</name>
      <email>jk@yahoo.com</email>
   </author>
</xml>
```

Figure 3.13: A data container with three XML documents as the result of a *join* operation ($D(\mathcal{G}) \bowtie_{\varphi} D(\mathcal{H})$)

3.5 XML Algebra Properties

XML algebra operators possess common associativity and distributivity properties as in the relational algebra model. Distributivity properties over the *union* operation of XML algebra and their proofs are as follows:

1. Projection: $\pi_F(D(\mathcal{G}) \cup D(\mathcal{H})) = \pi_F(D(\mathcal{G})) \cup \pi_F(D(\mathcal{H}))$;

   Proof. Let $x \in (D(\mathcal{G}) \cup D(\mathcal{H}))$.
   If $x \in (D(\mathcal{G}) \cup D(\mathcal{H}))$ then x is either in $D(\mathcal{G})$ or in $D(\mathcal{H})$.
   $x \in D(\mathcal{G})$ or $x \in D(\mathcal{H})$
   $\pi_F(D(\mathcal{G}) \cup D(\mathcal{H})) \Rightarrow \pi_F(D(\mathcal{G})) \cup \pi_F(D(\mathcal{H}))$ □

2. Selection: $\sigma_\varphi(D(\mathcal{G}) \cup D(\mathcal{H})) = \sigma_\varphi(D(\mathcal{G})) \cup \sigma_\varphi(D(\mathcal{H}))$;

   Proof. Let $x \in (D(\mathcal{G}) \cup D(\mathcal{H}))$.
   If $x \in (D(\mathcal{G}) \cup D(\mathcal{H}))$ then x is either in $D(\mathcal{G})$ or in $D(\mathcal{H})$.
   $x \in D(\mathcal{G})$ or $x \in D(\mathcal{H})$
   $\sigma_\varphi(D(\mathcal{G}) \cup D(\mathcal{H})) \Rightarrow \sigma_\varphi(D(\mathcal{G})) \cup \sigma_\varphi(D(\mathcal{H}))$ □

3. Union: $(D(\mathcal{G}) \cup D(\mathcal{H})) \cup D(\mathcal{K}) = D(\mathcal{G}) \cup (D(\mathcal{H}) \cup D(\mathcal{K})) = D(\mathcal{G}) \cup D(\mathcal{H}) \cup D(\mathcal{K}) = (D(\mathcal{G}) \cup D(\mathcal{K})) \cup (D(\mathcal{H}) \cup D(\mathcal{K}))$

   Proof. Let $x \in (D(\mathcal{G}) \cup D(\mathcal{H})) \cup D(\mathcal{K})$.
   If $x \in (D(\mathcal{G}) \cup D(\mathcal{H})) \cup D(\mathcal{K})$ then x is either in $(D(\mathcal{G}) \cup D(\mathcal{H}))$ or in $D(\mathcal{K})$.
   $x \in (D(\mathcal{G}) \cup D(\mathcal{H}))$ or $x \in D(\mathcal{K})$
   $x \in D(\mathcal{G})$ or $x \in D(\mathcal{H})$ or $x \in D(\mathcal{K})$
   Therefore:

   $$(D(\mathcal{G}) \cup D(\mathcal{H})) \cup D(\mathcal{K}) \Rightarrow D(\mathcal{G}) \cup D(\mathcal{H}) \cup D(\mathcal{K}) \tag{3.1}$$

   If $x \in (D(\mathcal{G}) \cup D(\mathcal{H})) \cup D(\mathcal{K})$ then x is either in $(D(\mathcal{G}) \cup D(\mathcal{H}))$ or in $D(\mathcal{K})$.
   $x \in (D(\mathcal{G}) \cup D(\mathcal{H}))$ or $x \in D(\mathcal{K})$
   $x \in D(\mathcal{G})$ or $x \in D(\mathcal{H})$ or $x \in D(\mathcal{K})$
   $x \in D(\mathcal{G})$ or $x \in (D(\mathcal{H})) \cup D(\mathcal{K}))$
   Therefore:

   $$(D(\mathcal{G}) \cup D(\mathcal{H})) \cup D(\mathcal{K}) \Rightarrow D(\mathcal{G}) \cup (D(\mathcal{H}) \cup D(\mathcal{K})) \tag{3.2}$$

If $x \in (D(\mathcal{G}) \cup D(\mathcal{H})) \cup D(\mathcal{K})$ then x is either in $(D(\mathcal{G}) \cup D(\mathcal{H}))$ or in $D(\mathcal{K})$.
$x \in (D(\mathcal{G}) \cup D(\mathcal{H}))$ or $x \in D(\mathcal{K})$
$x \in D(\mathcal{G})$ or $x \in D(\mathcal{H})$ or $x \in D(\mathcal{K})$
$x \in D(\mathcal{G})$ or $x \in D(\mathcal{K})$ or $x \in D(\mathcal{H})$ or $x \in D(\mathcal{K})$
$x \in D(\mathcal{G})$ or $x \in D(\mathcal{K})$ or $x \in (D(\mathcal{H})) \cup D(\mathcal{K}))$

$x \in (D(\mathcal{G}) \cup x \in D(\mathcal{K}))$ or $x \in (D(\mathcal{H})) \cup D(\mathcal{K}))$
Therefore:

$$(D(\mathcal{G}) \cup D(\mathcal{H})) \cup D(\mathcal{K}) \Rightarrow (D(\mathcal{G}) \cup D(\mathcal{K})) \cup (D(\mathcal{H}) \cup D(\mathcal{K})) \tag{3.3}$$

From (3.1), (3.2) and (3.3):
$(D(\mathcal{G}) \cup D(\mathcal{H})) \cup D(\mathcal{K}) = D(\mathcal{G}) \cup (D(\mathcal{H}) \cup D(\mathcal{K})) = D(\mathcal{G}) \cup D(\mathcal{H}) \cup D(\mathcal{K}) =$
$(D(\mathcal{G}) \cup D(\mathcal{K})) \cup (D(\mathcal{H}) \cup D(\mathcal{K}))$; □

4. Join: $(D(\mathcal{G}) \cup D(\mathcal{H})) \bowtie_\varphi D(\mathcal{K}) = (D(\mathcal{G}) \bowtie_\varphi D(\mathcal{K})) \cup (D(\mathcal{H}) \bowtie_\varphi D(\mathcal{K}))$;

Proof. Let $x \in (D(\mathcal{G}) \cup D(\mathcal{H})) \bowtie_\varphi D(\mathcal{K})$. If $x \in (D(\mathcal{G}) \cup D(\mathcal{H})) \bowtie_\varphi D(\mathcal{K})$, then x is either in $(D(\mathcal{G})$ or $D(\mathcal{H}))$ and in $D(\mathcal{K})$
$(x \in D(\mathcal{G})$ or $x \in D(\mathcal{H}))$ and $x \in D(\mathcal{K})$
$(x \in D(\mathcal{G})$ and $x \in D(\mathcal{K}))$ or $(x \in D(\mathcal{H})$ and $x \in D(\mathcal{K}))$
$x \in (D(\mathcal{G})$ and $D(\mathcal{K}))$ or $x \in (D(\mathcal{H})$ and $D(\mathcal{K}))$
$x \in (D(\mathcal{G}) \bowtie_\varphi D(\mathcal{K})) \cup (D(\mathcal{H}) \bowtie_\varphi D(\mathcal{K}))$
Therefore,
$(D(\mathcal{G}) \cup D(\mathcal{H})) \bowtie_\varphi D(\mathcal{K}) = (D(\mathcal{G}) \bowtie_\varphi D(\mathcal{K})) \cup (D(\mathcal{H}) \bowtie_\varphi D(\mathcal{K}))$ □

5. Antijoin: $(D(\mathcal{G}) \cup D(\mathcal{H})) \sim_\varphi D(\mathcal{K}) = (D(\mathcal{G}) \sim_\varphi D(\mathcal{K})) \cup (D(\mathcal{H}) \sim_\varphi D(\mathcal{K}))$

Proof. Let $x \in (D(\mathcal{G}) \cup D(\mathcal{H})) \sim_\varphi D(\mathcal{K})$. If $x \in (D(\mathcal{G}) \cup D(\mathcal{H})) \sim_\varphi D(\mathcal{K})$, then x is in $(D(\mathcal{G})$ or $D(\mathcal{H}))$, but not in $D(\mathcal{K})$
$(x \in D(\mathcal{G})$ or $x \in D(\mathcal{H}))$ and $x \notin D(\mathcal{K})$
$(x \in D(\mathcal{G})$ and $x \notin D(\mathcal{K}))$ or $(x \in D(\mathcal{H})$ and $x \notin D(\mathcal{K}))$
$x \in (D(\mathcal{G}) \sim_\varphi D(\mathcal{K})) \cup (D(\mathcal{H}) \sim_\varphi D(\mathcal{K}))$
Therefore,
$(D(\mathcal{G}) \cup D(\mathcal{H})) \sim_\varphi D(\mathcal{K}) = (D(\mathcal{G}) \sim_\varphi D(\mathcal{K})) \cup (D(\mathcal{H}) \sim_\varphi D(\mathcal{K}))$ □

6. Antijoin(2): $D(\mathcal{G}) \sim_\varphi (D(\mathcal{H}) \cup D(\mathcal{K})) = (D(\mathcal{G}) \sim_\varphi D(\mathcal{H})) \sim_\varphi D(\mathcal{K})$.

Proof. Let $x \in D(\mathcal{G}) \sim_\varphi (D(\mathcal{H}) \cup D(\mathcal{K}))$. If $x \in D(\mathcal{G}) \sim_\varphi (D(\mathcal{H}) \cup D(\mathcal{K}))$, then x is in $D(\mathcal{G})$, but neither in $D(\mathcal{H})$ nor $D(\mathcal{K})$
$x \in D(\mathcal{G})$ and $x \notin (D(\mathcal{H})$ or $D(\mathcal{K}))$
$x \in D(\mathcal{G})$ and $x \notin D(\mathcal{H})$ and $x \notin D(\mathcal{K})$
$(x \in D(\mathcal{G})$ and $x \notin D(\mathcal{H}))$ and $x \notin D(\mathcal{K})$
$x \in (D(\mathcal{G}) \sim_\varphi D(\mathcal{H}))$ and $x \notin D(\mathcal{K})$
$x \in (D(\mathcal{G}) \sim_\varphi D(\mathcal{H})) \sim_\varphi D(\mathcal{K})$
Therefore,
$D(\mathcal{G}) \sim_\varphi (D(\mathcal{H}) \cup D(\mathcal{K})) = (D(\mathcal{G}) \sim_\varphi D(\mathcal{H})) \sim_\varphi D(\mathcal{K})$ □

When data containers $D(\mathcal{G}), D(\mathcal{H})$, and $D(\mathcal{K})$ share common paths to satisfy a propositional condition of operations, then the distributivity of the *antijoin* operation over *join*, *union* and *antijoin* can be determined as follows:

7. $(D(\mathcal{G}) \sim_\varphi D(\mathcal{H})) \bowtie_\varphi D(\mathcal{K}) = ((D(\mathcal{G}) \bowtie_\varphi D(\mathcal{K})) \sim_\varphi (D(\mathcal{H}) \bowtie_\varphi D(\mathcal{K}))$;

   Proof. Let $x \in (D(\mathcal{G}) \sim_\varphi D(\mathcal{H})) \bowtie_\varphi D(\mathcal{K})$.
   If $x \in (D(\mathcal{G}) \sim_\varphi D(\mathcal{H})) \bowtie_\varphi D(\mathcal{K})$, then x is in $D(\mathcal{G})$ and not in $D(\mathcal{H})$, and x is in $D(\mathcal{K})$
   $(x \in D(\mathcal{G})$ and $x \notin D(\mathcal{H}))$ and $x \in D(\mathcal{K})$
   $x \in D(\mathcal{G})$ and $x \in D(\mathcal{K})$ and $x \notin D(\mathcal{H})$
   $(x \in D(\mathcal{G})$ and $x \in D(\mathcal{K}))$ and $x \notin D(\mathcal{H})$
   $x \in (D(\mathcal{G})$ and $D(\mathcal{K}))$ and $x \notin D(\mathcal{H})$
   $x \in (D(\mathcal{G}) \bowtie_\varphi D(\mathcal{K}))) \sim_\varphi D(\mathcal{H})$
   Therefore,
   $(D(\mathcal{G}) \sim_\varphi D(\mathcal{H}))) \bowtie_\varphi D(\mathcal{K}) = (D(\mathcal{G}) \bowtie_\varphi D(\mathcal{K})) \sim_\varphi (D(\mathcal{H}) \bowtie_\varphi D(\mathcal{K}))$ □

8. $D(\mathcal{G}) \bowtie_\varphi (D(\mathcal{H}) \sim_\varphi D(\mathcal{K})) = (D(\mathcal{G}) \bowtie_\varphi D(\mathcal{H})) \sim_\varphi D(\mathcal{K})$;

   Proof. Let $x \in (D(\mathcal{G}) \bowtie_\varphi (D(\mathcal{H}) \sim_\varphi D(\mathcal{K})))$.
   If $x \in (D(\mathcal{G}) \bowtie_\varphi (D(\mathcal{H}) \sim_\varphi D(\mathcal{K})))$, then x is in $D(\mathcal{G})$ and in $(D(\mathcal{H}) \sim_\varphi D(\mathcal{K}))$
   $x \in D(\mathcal{G})$ and $(x \in D(\mathcal{H})$ and $x \notin D(\mathcal{K}))$
   $x \in D(\mathcal{G})$ and $x \in D(\mathcal{H})$ and $x \notin D(\mathcal{K})$
   $(x \in D(\mathcal{G})$ and $x \in D(\mathcal{H}))$ and $x \notin D(\mathcal{K})$
   $x \in (D(\mathcal{G})$ and $D(\mathcal{H}))$ and $x \notin D(\mathcal{K})$
   $x \in (D(\mathcal{G}) \bowtie_\varphi D(\mathcal{H})) \sim_\varphi D(\mathcal{K})$
   Therefore,
   $D(\mathcal{G}) \bowtie_\varphi (D(\mathcal{H}) \sim_\varphi D(\mathcal{H})) = (D(\mathcal{G}) \bowtie_\varphi D(\mathcal{H})) \sim_\varphi D(\mathcal{K})$ □

9. $(D(\mathcal{G}) \sim_\varphi D(\mathcal{H})) \cup D(\mathcal{K}) = (D(\mathcal{G}) \cup D(\mathcal{K})) \sim_\varphi (D(\mathcal{H}) \sim_\varphi D(\mathcal{K}))$;

   Proof. Let $D(\mathcal{G}) \in (D(\mathcal{G}) \sim_\varphi D(\mathcal{H})) \cup D(\mathcal{K})$.
   If $x \in (D(\mathcal{G}) \sim_\varphi D(\mathcal{H})) \cup D(\mathcal{K})$, then x is in $(D(\mathcal{G}) \sim_\varphi D(\mathcal{H}))$ or x is in $D(\mathcal{K})$
   $(x \in D(\mathcal{G})$ and $x \notin D(\mathcal{H}))$ or $x \in D(\mathcal{K})$
   $(x \in D(\mathcal{G})$ or $x \in D(\mathcal{K}))$ and $(x \notin D(\mathcal{H})$ or $x \in D(\mathcal{K}))$
   $x \in (D(\mathcal{G})$ or $D(\mathcal{K}))$ and $\neg(x \in D(\mathcal{H})$ and $x \notin D(\mathcal{K}))$
   $(D(\mathcal{G}) \cup D(\mathcal{K})) \sim_\varphi (D(\mathcal{H}) \sim_\varphi D(\mathcal{K}))$
   Therefore,
   $(D(\mathcal{G}) \sim_\varphi D(\mathcal{H})) \cup D(\mathcal{K}) = (D(\mathcal{G}) \cup D(\mathcal{K})) \sim_\varphi (D(\mathcal{H}) \sim_\varphi D(\mathcal{K}))$; □

10. $D(\mathcal{G}) \cup (D(\mathcal{H}) \sim_\varphi D(\mathcal{K})) = (D(\mathcal{G}) \cup D(\mathcal{H})) \sim_\varphi (D(\mathcal{K}) \sim_\varphi D(\mathcal{G}))$;

    Proof. Let $x \in (D(\mathcal{G}) \cup (D(\mathcal{H}) \sim_\varphi D(\mathcal{K})))$.
    If $x \in (D(\mathcal{G}) \cup (D(\mathcal{H}) \sim_\varphi D(\mathcal{K})))$, then x is in $D(\mathcal{G})$ or x in $(D(\mathcal{H}) \sim_\varphi D(\mathcal{K}))$
    $x \in D(\mathcal{G})$ or $(x \in D(\mathcal{H})$ and $x \notin D(\mathcal{K}))$
    $(x \in D(\mathcal{G})$ or $x \in D(\mathcal{H}))$ and $(x \in D(\mathcal{G})$ or $x \notin D(\mathcal{K}))$
    $x \in (D(\mathcal{G})$ or $D(\mathcal{H}))$ and $\neg(x \notin D(\mathcal{G})$ and $x \in D(\mathcal{K}))$
    $x \in (D(\mathcal{G})$ or $D(\mathcal{H}))$ and $\neg(x \in D(\mathcal{K})$ and $x \notin D(\mathcal{G}))$
    $x \in (D(\mathcal{G})$ or $D(\mathcal{H}))$ and $\neg(x \in D(\mathcal{K})$ and $x \notin D(\mathcal{G}))$
    Therefore,
    $D(\mathcal{G}) \cup (D(\mathcal{H}) \sim_\varphi D(\mathcal{K})) = (D(\mathcal{G}) \cup D(\mathcal{H})) \sim_\varphi (D(\mathcal{K}) \sim_\varphi D(\mathcal{G}))$; □

11. $(D(\mathcal{G}) \sim_\varphi D(\mathcal{H})) \sim_\varphi D(\mathcal{K}) = D(\mathcal{G}) \sim_\varphi D(\mathcal{H}) \sim_\varphi D(\mathcal{K})$;

    Proof. Let $x \in (D(\mathcal{G}) \sim_\varphi D(\mathcal{H})) \sim_\varphi D(\mathcal{K})$.
    If $x \in (D(\mathcal{G}) \sim_\varphi D(\mathcal{H})) \sim_\varphi D(\mathcal{K})$, then x is in $(D(\mathcal{G}) \sim_\varphi D(\mathcal{H}))$ and x is not in $D(\mathcal{K})$
    $x \in D(\mathcal{G})$ and $x \notin D(\mathcal{H})$ and $x \notin D(\mathcal{K})$
    Therefore,
    $(D(\mathcal{G}) \sim_\varphi D(\mathcal{H})) \sim_\varphi D(\mathcal{K}) = D(\mathcal{G}) \sim_\varphi D(\mathcal{H}) \sim_\varphi D(\mathcal{K})$; □

12. $D(\mathcal{G}) \sim_\varphi (D(\mathcal{H}) \sim_\varphi D(\mathcal{K})) = (D(\mathcal{G}) \sim_\varphi D(\mathcal{H})) \cup (D(\mathcal{G}) \bowtie_\varphi D(\mathcal{K}))$

    Proof. Let $x \in (D(\mathcal{G}) \sim_\varphi (D(\mathcal{H}) \sim_\varphi D(\mathcal{K})))$.
    If $x \in (D(\mathcal{G}) \sim_\varphi (D(\mathcal{H}) \sim_\varphi D(\mathcal{K})))$, then x is in $D(\mathcal{G})$ and x not in $(D(\mathcal{H}) \sim_\varphi D(\mathcal{K}))$

$x \in D(\mathcal{G})$ and not ($x \in D(\mathcal{H})$ and $x \notin D(\mathcal{K})$)
$x \in D(\mathcal{G})$ and ($x \notin D(\mathcal{H})$ or $x \in D(\mathcal{K})$)
($x \in D(\mathcal{G})$ and $x \notin D(\mathcal{H})$) or ($x \in D(\mathcal{G})$ and $x \in D(\mathcal{K})$)
Therefore,
$D(\mathcal{G}) \sim_\varphi (D(\mathcal{H}) \sim_\varphi D(\mathcal{K})) = (D(\mathcal{G}) \sim_\varphi D(\mathcal{H})) \cup (D(\mathcal{G}) \ltimes_\varphi D(\mathcal{K}))$; □

The properties listed above play an important role in enabling incremental processing in an online data integration system.

3.6 XML Algebra vs Relational Algebra

The XML algebra described earlier has an important property when compared with relational algebra. If we simplify a data container to have a single set of ETG as its schema, we restrict all ETGs to contain production rules with a simple sequence of elements, and we simplify each document in a data container such that it represents a row in a relational table, and then the system of operations reduces to relational algebra.

Example 3.30. *Consistency of XML algebra to relational algebra*
Let $D(\{G\})$ be a data container with a single set schema, G=(N,T,A,S,P). The ETG is as follows:

```
N={S,ISBN,TITLE AUT_EMAIL,SUBJECT,ED_EMAIL}
T={xml,isbn,title,aut_email,subject,ed_email}
A={id}
P={S→xml[id](ISBN,TITLE AUT_EMAIL SUBJECT ED_EMAIL),
   ISBN→isbn,TITLE→title,AUT_EMAIL→aut_email,
   SUBJECT→subject,ED_EMAIL→ed_email}
```

The above ETG defines the structure of a `book` document with one level depth. Based on the ETG, we are able to generate a `book` XML document, as in Figure 3.14

Let $D(\mathcal{H})$ be a data container with a single set schema, H=(N,T,A,S,P). The ETG is as follows:

```
N={S,NAME,EMAIL}
T={xml,name,email}
A={id}
P={S→xml[id](NAME,EMAIL),NAME→name,EMAIL→email}
```

```
<xml id="B01">
   <isbn>9872347765</isbn>
   <title>XML</title>
   <aut_email>andy@yahoo.com</aut_email>
   <subject>Data</subject>
   <ed_email>ross@gmail.com</ed_email>
</xml>
```

Figure 3.14: A `book` XML document with a simple structure

The above ETG describes the structure of an `author` document with one level depth. Based on the above ETG, we can generate an XML document as in Figure 3.15

```
<xml id="A01">
   <name>Andy Cole</title>
   <email>andy@yahoo.com</aut_email>
</xml>

<xml id="A02">
   <name>Ben Johnson</name>
   <email>ben@yahoo.com</email>
</xml>
```

Figure 3.15: Two `author` XML documents with a simple structure

The same ETG for the `author` document can be used to define the `editor` document. Then, we can generate an XML document, as in Figure 3.16

```
<xml id="E01">
       <name>Ross Marrie</name>
   <email>ross@gmail.com</email>
</xml>
```

Figure 3.16: An `editor` XML document with a simple structure

The four documents in Figure 3.14 and 3.15 represent rows in the relational tables, where the `xml` element and its `id` attribute represents the *row id* in a relational table, whereas a pair of opening and closing *tags* represents a *field name*.

Let $D(\mathcal{G})$ be a data container with a single XML document (Figure 3.14); Figure 3.15 shows two XML documents in a data container $D(\mathcal{H})$. Figure 3.16 shows an XML document in a data container $D(\mathcal{K})$, where $\mathcal{K}$=$\{$H$\}$. Data containers are a collection of XML documents, therefore they represent tables in the relational model as shown in Figure 3.17.

	isbn	title	aut_email	subject	ed_email
	9872347765	XML	andy@yahoo.com	Data	ross@gmail.com
▶*	NULL	NULL	NULL	NULL	NULL

(a)

	name	email
	Andy Cole	andy@yahoo.com
	Ben Johnson	ben@yahoo.com
▶*	NULL	NULL

(b)

	name	email
	Ross Marrie	ross@gmail.com
▶*	NULL	NULL

(c)

Figure 3.17: (a) `Book` table which is compatible with data container $D(\mathcal{G})$, (b) `author` table which corresponds to data container $D(\mathcal{H})$, and (c) `editor` table which is compatible with data container $D(\mathcal{K})$

□

Next, XML algebra operations is performed over data containers $D(\mathcal{G})$ and $D(\mathcal{H})$ as follows:

1. Consider a *selection* operation $\sigma_{\varphi}(D(\mathcal{H}))$, where φ=`//name="Andy Cole"`.

   The operation returns a data container with schema $\mathcal{F}$, where $\mathcal{F} = \mathcal{H}$. $D(\mathcal{F})$ contains an XML document:

```
<xml id="S01">
   <name>Andy Cole</title>
   <email>andy@yahoo.com</aut_email>
</xml>
```

   The result is consistent with the *selection* operation in relational algebra which returns a new table with a corresponding row result and with a new *row id*.

   The difference is that when the *selection* operation returns no data, relational algebra produces a table with a structure with no rows in it. In contrast,

selection in XML algebra returns a data container with no XML document, and has an empty set of ETG as its schema, $D(\mathcal{F})$ where $\mathcal{F} = \emptyset$.

2. Consider a *projection* operation $\pi_H(D(\mathcal{G}))$, where H is an ETG as follows:

```
N={S,TITLE,SUBJECT}
T={xml,title,subject}
A={id,isbn,lang}
P={S→xml[id](TITLE SUBJECT),
   TITLE→title,SUBJECT→subject}
```

   The *projection* operation produces a data container of XML documents $D(\mathcal{F})$, where $\mathcal{F} = \{H\}$. The XML document results have a new structure, as defined in H. Meanwhile, the *projection* operation on the relational model produces a table with a new structure as in the projection argument. It concludes that the result of the *projection* operation in our XML algebra is consistent with that in relational algebra.

   Similar to the *selection* operation, when the *projection* operation in XML algebra returns no document, a data container with an empty set of ETGs is returned, such that $D(\mathcal{F})$ where $\mathcal{F} = \emptyset$.

3. *Union* operation over $D(\mathcal{H})$ and $D(\mathcal{K})$ produces a data container $D(\mathcal{F})$. Since $\mathcal{H} = \mathcal{K}$, then the resulting data container is $D(\mathcal{F})$, where $\mathcal{F} = \mathcal{H} = \mathcal{K}$. $D(\mathcal{F})$ contains three XML documents with the same structure, namely two `author` documents and one `editor` document.

   In relational algebra, equivalent results are obtained when a *union* operation is performed on `author` and `editor` tables. Since they have identical structures, we obtain a new table with a single structure which contains three rows.

   The XML algebra has an advantage that a *union* operation can be performed on two data containers with different schemas. In other words, the *union* operation in XML algebra allows unification of XML documents with different structures in a data container result. Then, the schema of the data container result is obtained by unification of all schemas from both data containers.

4. Consider a *join* operation $D(\mathcal{G}) \bowtie_\varphi D(\mathcal{H})$, where φ="`//aut_email=//email`", `//aut_email` is a path in $x(m) \in D(\mathcal{G})$ and `//email` is a path in $y(n) \in D(\mathcal{H})$. In XML algebra, computation of the *join* operation is performed

such that the data container result has a schema {G+H:G$\in \mathcal{G}$,$H \in \mathcal{H}$}. The *merge* operation G+H results in an ETG F as follows:

```
N={S,ISBN,TITLE AUT_EMAIL,SUBJECT,ED_EMAIL,NAME,EMAIL}
T={xml,isbn,title,aut_email,subject,ed_email,name,email}
A={id}
P={S→xml[id](ISBN,TITLE AUT_EMAIL SUBJECT ED_EMAIL NAME EMAIL),
   ISBN→isbn,TITLE→title,AUT_EMAIL→aut_email,
   SUBJECT→subject,ED_EMAIL→ed_email,NAME→name,EMAIL→email}
```

The data container result consists of an XML document as shown in Figure 3.18.

```
<xml id="J01">
   <isbn>9872347765</isbn>
   <title>XML</title>
   <aut_email>andy@yahoo.com</aut_email>
   <subject>Data</subject>
   <ed_email>ross@gmail.com</ed_email>
   <name>Andy Cole</name>
   <email>andy@yahoo.com</email>
</xml>
```

Figure 3.18: A `book` XML document with a simple structure

In relational algebra, a *join* operation can be performed by a query:

```
SELECT * from Book, Author WHERE Book.aut_email=Author.email
```

The result of the *join* operation is shown in Figure 3.19.

Results | Messages

	isbn	title	aut_email	subject	ed_email	name	email
1	9872347765	XML	andy@yahoo.com	Data	ross@gmail.com	Andy Cole	andy@yahoo.com

Figure 3.19: Join result in the relational algebra

We can see that both results are compatible, therefore a *join* operation in XML algebra is consistent with the one in the relational algebra.

5. The last operation is the *antijoin* operation. An example is $D(\mathcal{G}) \sim_{\varphi} D(\mathcal{H})$ where φ=`book//aut_email=author/email`. `book//aut_email` is a path in $x(m) \in D(\mathcal{G})$ and `author/email` is a path in $y(n) \in D(\mathcal{H})$.

   In this example, an *antijoin* operation results in no XML document. Therefore, the data container result has a schema $\mathcal{F} = \emptyset$.

In relational algebra, the operation can be written as: `select * FROM Book WHERE aut_email not IN (SELECT email FROM Author)` and results an empty table. The structure of the result table is the same as the structure of `book` table.

Let U be a set of all data containers, and $V \subseteq S$ be a set of data containers such that there exists a transformation t of any $D(G) \in V$ into a relational table $R \in W$. Figure 3.20 shows a transformation function t such that an equivalence mapping from V to W exists.

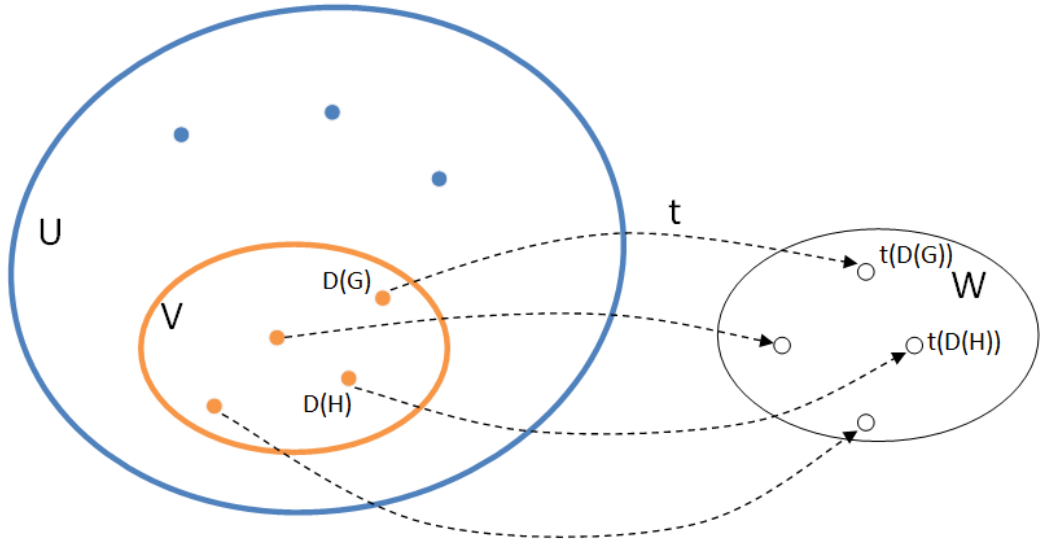

Figure 3.20: Equivalent mapping from the XML model to the relational model

Let θ be an operation on two data containers in V, such that $\theta(D(\mathcal{G}), D(\mathcal{H}))$ produces a data container as its result.

Theorem 1. *For all $D(\mathcal{G}), D(\mathcal{H}) \in V$ and any operation θ in XML algebra where $\theta(D(\mathcal{G}), D(\mathcal{H})) \in V$, an equivalent operation Θ exists in relational algebra such that $t(\theta(D(\mathcal{G}), D(\mathcal{H}))) = t(D(\mathcal{G}))\ \Theta\ t(D(\mathcal{H}))$.*

Proof. Let we have the following:

1. $G = (N_G, T_G, A_G, S_G, P_G), H = (N_H, T_H, A_H, S_H, P_H)$ be ETGs.
2. $m \in T_G*$ be a sentence of G, and $n \in T_H*$ be a sentence of H.
3. $X = T_G - \{\texttt{xml}\}$ and $Y = T_H - \{\texttt{xml}\}$ be sets of terminal symbols without a terminal symbol `xml`. $X \cap Y = \emptyset$.
4. $x(m) \in D(\mathcal{G}), y(n) \in D(\mathcal{H})$ be XML documents which are equivalent to relational rows, where $m =$`xml[id]`(m'), $n =$`xml[id]`(n').

5. `xml[id]` is equivalent to a row id in the relational table.

6. $m' \in X*$, $n' \in Y*$ be sets of unique terminal symbols.

7. α be a transformation function of an XML document into a particular row in a relational table, and is defined as $\alpha(x(m)) = \{\texttt{rowID}, \langle c = value\rangle : \texttt{xml[id]} \rightarrow \texttt{rowID}, \forall t \in m' \rightarrow c, \texttt{t/text()} \rightarrow value\}$, where c represents a column in a relational table. Therefore, $\alpha(x(m)) = r$.

8. r be a row in a relational table, `rowID` be a row identity, $C = \{c_1, \ldots, c_n\}$ be a set of columns in r, and β be a transformation function of r into an XML document $x(m)$, and is defined as $\beta(r) = \{x(\texttt{xml[id]}(m')) : \texttt{rowID} \rightarrow \texttt{xml[id]}, \forall c \in C \rightarrow t \in m'\}$. Therefore, $\beta(r) = x(m)$

Therefore, $t(D(\mathcal{G})) = \{\alpha(x(m)) : x(m) \in D(\mathcal{G})\}$ represents transformation of an XML data container into a table in a relational model. A table $R = \{r_1, \ldots, r_n\}$ is a set of rows where $r_i = \alpha(x_i(m))$. Therefore, $t(D(\mathcal{G})) = R$.

Definition 27. *Let c be a column in a relational table, and v be a string value. Let θ be an operator in a set of comparison operator $\{=, <>, <, >, <=, >=\}$. A condition expression in the relational model is denoted as Φ, and is defined as a propositional formula that consists of proposition(s) in the form of $(c\ \theta\ c)$ or $(c\ \theta\ v)$, and the logical operators $\wedge$ (and), $\vee$ (or) and $\neg$ (negation).*

1. The selection operation $\sigma_\varphi(D(\mathcal{G})) = \{x(m) : x(m) \in D(\mathcal{G})\ and\ f(x(m), \varphi) = true\}$
   Let Σ be a selection operation in the relational model.

$$\begin{aligned} t(\sigma_\varphi(D(\mathcal{G}))) =& t(\{x(m) : x(m) \in D(\mathcal{G})\ and\ f(x(m), \varphi) = true\}) \\ =& \{\alpha(x(m)) : x(m) \in D(\mathcal{G})\ and\ f(x(m), \varphi) = true\} \end{aligned} \tag{3.4}$$

   Let $f(x(m), \varphi) \equiv eval(r, \Phi)$. The selection operation on $t(D(\mathcal{G}))$ in the domain of W is as follows:

$$\begin{aligned} \Sigma_\Phi(t(D(\mathcal{G}))) =& \Sigma_\Phi(R) \\ =& \{r : r \in R\ and\ eval(r, \Phi) = true\} \\ =& \{\alpha(x(m)) : x(m) \in D(\mathcal{G})\ and\ f(x(m), \varphi) = true\} \end{aligned} \tag{3.5}$$

   From (3.4) and (3.5), $t(\sigma_\varphi(D(\mathcal{G}))) = \Sigma_\Phi(t(D(\mathcal{G})))$. Therefore, we conclude that $\sigma_\varphi \equiv \Sigma_\Phi$.

2. The projection operation $\pi_H(D(\mathcal{G})) = \{y(m') : \exists x(m) \in D(\mathcal{G}), y(m') = x(m)[H]\}$. Let Π be a projection operation in the relational model.

$$\begin{aligned} t(\pi_H(D(\mathcal{G}))) =& t(\{y(m') : \exists x(m) \in D(\mathcal{G}), y(m') = x(m)[H]\}) \\ =& \{\alpha(y(m')) : \exists x(m) \in D(\mathcal{G}), y(m') = x(m)[H]\} \end{aligned} \tag{3.6}$$

Let $\mathcal{G} = \{G\}$, and $H \sqsubseteq G$. Terminal symbol $T_H \equiv C = \{c_1, \ldots, c_n\}$, $x(m)[H] = y(m')$, and $\alpha(x(m)[H]) \equiv r[c_1, \ldots, c_n]$. The projection operation on $t(D(\mathcal{G}))$ in the domain of W is as follows:

$$\begin{aligned} \Pi_{c_1,\ldots,c_n}(t(D(\mathcal{G}))) =& \Pi_{c_1,\ldots,c_n}(R) \\ =& \{r[c_1, \ldots, c_n] : \exists r \in R\} \\ =& \{\alpha(x(m)[H]) : \exists x(m) \in D(\mathcal{G})\} \\ =& \{\alpha(y(m')) : \exists x(m) \in D(\mathcal{G}), y(m') = x(m)[H]\} \end{aligned} \tag{3.7}$$

From (3.6) and (3.7), $t(\pi_H(D(\mathcal{G}))) = \Pi_{c_1,\ldots,c_n}(t(D(\mathcal{G})))$. Therefore, we conclude that $\pi_H \equiv \Pi_{[c_1,\ldots,c_n]}$.

3. The join operation $D(\mathcal{G}) \bowtie_\varphi D(\mathcal{H}) = \{z(o) : \exists x(m) \in D(\mathcal{G}), y(n) \in D(\mathcal{H}), z(o) = x(m) \bullet_\varphi y(n)\}$.
Let $\bowtie_\Phi$ be a *join* operation in the relational model.

$$\begin{aligned} & t(D(\mathcal{G}) \bowtie_\varphi D(\mathcal{H})) = \\ & = t(\{z(o) : \exists x(m) \in D(\mathcal{G}), y(n) \in D(\mathcal{H}), z(o) = x(m) \bullet_\varphi y(n)\}) \\ & = \{\alpha(z(o)) : \exists x(m) \in D(\mathcal{G}), y(n) \in D(\mathcal{H}), z(o) = x(m) \bullet_\varphi y(n)\} \\ & = t\{\alpha(x(m) \bullet_\varphi y(n)) : \exists x(m) \in D(\mathcal{G}), y(n) \in D(\mathcal{H})\} \end{aligned} \tag{3.8}$$

Let $t(D(\mathcal{G})) = R$, $t(D(\mathcal{H})) = S$, $\varphi \equiv \Phi$, and $r \times s \equiv x(m) \bullet_\varphi y(n)$.

$$\begin{aligned} t(D(\mathcal{G})) \bowtie_\Phi t(D(\mathcal{H})) =& R \bowtie_\Phi S \\ =& \{r \times s : r \in R \textit{ and } s \in S \textit{ and } eval((r \times s), \Phi) = true\} \\ =& \{\alpha(x(m) \bullet_\varphi y(n)) : \exists x(m) \in D(\mathcal{G}), y(n) \in D(\mathcal{H})\} \end{aligned} \tag{3.9}$$

From (3.8) and (3.9), $t(D(\mathcal{G}) \bowtie_\varphi D(\mathcal{H})) = (t(D(\mathcal{G})) \bowtie_\Phi t(D(\mathcal{G})))$. Therefore, we can conclude that $\bowtie_\varphi \equiv \bowtie_\Phi$.

4. The union operation $D(\mathcal{G}) \cup D(\mathcal{H}) = \{x(m) : x(m) \in D(\mathcal{G}) \text{ or } x(m) \in D(\mathcal{H})\}$.
   Let $\Cup$ be a union operation in the relational model.

$$\begin{aligned} t(D(\mathcal{G}) \cup D(\mathcal{H})) =& t(\{x(m) : x(m) \in D(\mathcal{G}) \text{ or } x(m) \in D(\mathcal{H})\}) \\ =& \{\alpha(x(m)) : x(m) \in D(\mathcal{G}) \text{ or } x(m) \in D(\mathcal{H})\} \end{aligned} \tag{3.10}$$

   Let $t(D(\mathcal{G})) = R$, $t(D(\mathcal{H})) = S$.

$$\begin{aligned} t(D(\mathcal{G})) \Cup t(D(\mathcal{H})) =& R \Cup S \\ =& \{r : r \in R \text{ or } r \in S\} \\ =& \{\alpha(x(m)) : x(m) \in D(\mathcal{G}) \text{ or } x(m) \in D(\mathcal{H})\} \end{aligned} \tag{3.11}$$

   From (3.10) and (3.11), $t(D(\mathcal{G}) \cup D(\mathcal{H})) = (t(D(\mathcal{G})) \Cup t(D(\mathcal{G})))$. Therefore, we draw a conclusion that $cup \equiv \Cup$.

5. The antijoin operation $D(\mathcal{G}) \sim_\varphi D(\mathcal{H}) = \{x(m) : x(m) \in D(\mathcal{G}) \text{ and } \neg\exists y(n) \in D(\mathcal{H})(x(m) \bullet_\varphi y(n))\}$.
   Let $\triangleright_\Phi$ be an antijoin operation in the relational model.

$$\begin{aligned} & t(D(\mathcal{G}) \sim_\varphi D(\mathcal{H})) = \\ & = t(\{x(m) : x(m) \in D(\mathcal{G}) \text{ and } \neg\exists y(n) \in D(\mathcal{H})(x(m) \bullet_\varphi y(n))\}) \\ & = \{\alpha(x(m)) : x(m) \in D(\mathcal{G}) \text{ and } \neg\exists y(n) \in D(\mathcal{H})(x(m) \bullet_\varphi y(n))\} \end{aligned} \tag{3.12}$$

   Let $t(D(\mathcal{G})) = R$, $t(D(\mathcal{H})) = S$, $\varphi \equiv \Phi$, and $r \times s \equiv x(m) \bullet_\varphi y(n)$.

$$\begin{aligned} & t(D(\mathcal{G})) \triangleright_\Phi t(D(\mathcal{H})) = \\ & = R \triangleright_\Phi S \\ & = \{r : r \in R \text{ and } \neg\exists s \in S(r \times_\Phi s)\} \\ & = \{\alpha(x(m)) : x(m) \in D(\mathcal{G}) \text{ and } \neg\exists y(n) \in D(\mathcal{H})(x(m) \bullet_\varphi y(n))\} \end{aligned} \tag{3.13}$$

   From (3.12) and (3.13), $t(D(\mathcal{G}) \sim_\varphi D(\mathcal{H})) = (t(D(\mathcal{G})) \triangleright_\Phi t(D(\mathcal{G})))$. Therefore, we draw the conclusion that $\sim_\varphi \equiv \triangleright_\Phi$.

□

In conclusion, by using a simple data structure as a schema of data containers $D(\mathcal{G})$, XML algebra operations return consistent results with relational algebraic

operations.

In reality some data containers are not equivalent to those in the relational model, and shown as a set of U-V in Figure 3.20. These data containers may have the following special characteristics:

1. they have nested structures. In the relational model, nested structures are accommodated by a nested relational model.

2. their elements may have mixed values (child element, literal value, and/or attributes)

Accordingly, we investigate whether these special characteristics are acceptable in the proposed XML algebra:
Let $D(\mathcal{G}'), D(\mathcal{H}')$ be any data containers. Let $D(\mathcal{G}') = \{x(m)\}$ where m=`xml[id="v1"]m'` is the structure of document $y(n)$, and $D(\mathcal{H}') = \{y(n)\}$ where n=`xml[id="v2"]n'` is the structure of document $x(m)$. Let `xml[id]` be the top structure of document to store document's identity (`val1, val2`). Let `m'`, `n'`$\in \{T \cup A\}*$ be any sequence of terminal (T) and attribute (A) symbols which represent the structure of XML documents without identities, and include XML structures equivalent to relational tables.

1. The *selection* operation ($\sigma_{\varphi}(D(\mathcal{G}'))$)
   The *selection* operation does not change the structure of documents (`m'`). It operates on every XML document in a data container which satisfies the condition. The condition expression is based on path expression. The *selection* operation on a $D(\mathcal{G}')$ returns a data container $D(\mathcal{G}') = \{x(m)\}$, where `m=xml[id="val1'"](m')`. Therefore with any structure of XML document, the *selection* operation is applicable to these data containers.

2. The *union* operation ($D(\mathcal{G}') \cup D(\mathcal{H}')$)
   Similar to *selection*, the *union* operation does not change the structure of documents. Operation ($D(\mathcal{G}') \cup D(\mathcal{H}')$) returns a data container $D(\mathcal{F}) = \{x(m), y(n)\}$ where $\mathcal{F} = \mathcal{G}' \cup \mathcal{H}'$. Therefore with any structure of XML document, the *union* operation is applicable.

3. The *join* operation ($D(\mathcal{G}') \bowtie_{\varphi} D(\mathcal{H}')$)
   $D(\mathcal{G}') \bowtie_{\varphi} D(\mathcal{H}')$ returns a data container $D(\mathcal{F}) = \{z(o)\}$ if a condition expression φ is satisfied. The result data container has a schema $\mathcal{F} = \{G' + H'\}$. A new XML document has a structure o=`xml[id="val3"](m'␣n')`, where `"val3"` is a new identity. As we can see from this result, for any

structures represented as `m'` and `n'` including the ones which equivalent to relational table, the *join* operation is applicable.

4. The *antijoin* operation ($D(\mathcal{G}') \sim_{\varphi} D(\mathcal{H}')$)
   $D(\mathcal{G}') \sim_{\varphi} D(\mathcal{H}')$ returns a data container $D(\mathcal{F}) = \{x(m)\}$ if a condition expression φ is not satisfied. However, $x(m)$ are assigned to a new identity value, where m=`xml[id="val3"](m')`. Therefore, the *antijoin* operation is applicable to any data containers including data containers which are equivalent to relational tables.

5. The projection operation ($\pi_H(D(\mathcal{G}'))$)
   $\pi_H(D(\mathcal{G}'))$ returns a data container $D(\mathcal{F}) = \{x(o)\}$, where o =`xml[id="val3"](m")`, $\mathcal{F} \subseteq \mathcal{G}'$, and `m"` is a substring of `m'`. Then, for any structure m', the *projection* operation is applicable.

Since XML and relational algebras are consistent, the properties of relational algebra can be applied to the XML algebra as follows:

1. The *selection* operation is idempotent and commutative:
   a. $\sigma_{\varphi_1}\sigma_{\varphi_2}(D(\mathcal{G}) = \sigma_{\varphi_2}\sigma_{\varphi_1}(D(\mathcal{G})$

2. A *selection* with complex condition expression can be broken down into:
   a. $\sigma_{\varphi_1 \wedge \varphi_2}(D(\mathcal{G})) = \sigma_{\varphi_1}(\sigma_{\varphi_2}(D(\mathcal{G}))) = \sigma_{\varphi_2}(\sigma_{\varphi_1}(D(\mathcal{G})))$
   b. $\sigma_{\varphi_1 \vee \varphi_2}(D(\mathcal{G})) = \sigma_{\varphi_1}(D(\mathcal{G})) \cup \sigma_{\varphi_2}(D(\mathcal{G}))$

3. The *projection* operation is idempotent:
   a. $\pi_K(\pi_H(D(\mathcal{G}))) = \pi_K(D(\mathcal{G}))$, where $K \sqsubseteq H$.

4. The *join* operation is commutative:
   a. $D(\mathcal{G}) \bowtie_{\varphi} D(\mathcal{H}) = D(\mathcal{H}) \bowtie_{\varphi} D(\mathcal{G})$

5. Distributive properties:
   a. $\sigma_{\varphi}(D(\mathcal{G}) \cup D(\mathcal{H})) = \sigma_{\varphi}(D(\mathcal{G})) \cup \sigma_{\varphi}(D(\mathcal{H}))$
   b. $(D(\mathcal{G}) \cup D(\mathcal{H})) \bowtie_{\varphi} D(\mathcal{K}) = (D(\mathcal{G}) \bowtie_{\varphi} D(\mathcal{K})) \cup (D(\mathcal{H}) \bowtie_{\varphi} D(\mathcal{K}))$

Consistency with relational algebra is necessary to gain advantages of existing research results which have been undertaken in relational algebra. Since it has been established and widely used, there are a number of performance tuning algorithms for relational algebra that can be applied to the XML algebra outlined in this chapter.

Chapter 4

Online Data Integration System

Online data integration in a distributed multi-database system is a continuous consolidation process of the data transmitted over a network with the data already available at a central site. The intermediate results of online data integration provide a user with the most up-to-date results of a query being processed by the system. Online integration applies an online processing where the units of data increments are instantly processed without having an entire set of data available. Then, the result of *incremental* data processing is combined with the current state to obtain a new processing state.

This chapter presents the core of this thesis: an online data integration system. An architecture and a framework as the backbone of the online data integration system is described in Section 4.1. An example of a *global query expression* is given in Section 4.2 to give a better understanding as to how the system works. Section 4.3 describes a decomposition strateqy to balance computation between the central and remote sites. Section 4.4 describes an algorithm to transform a *global query expression* into a *data integration expression*, and in Section 4.4.1 transformation of a *data integration expression* into *increment expressions* is presented. Section 4.4.2 describes an algorithm to generate an *online integration plan* based on the *increment expressions* generated earlier. Section 4.5 covers a *dynamic scheduling system* to efficiently manage execution of online integration plans.

4.1 Online Data Integration Architecture

Processing of data integration procedures starts when a user query arrives at a central site. The central site decomposes the user query into several sub-queries and sends them to the remote sites for computation. Then, the remote sites send the computation results back to the central site for further integration processing. Generally, the central site starts the integration process when all data become available at the central site.

In an online data integration system, the basic concepts of data integration systems are preserved, but the integration process at the central site is started as soon as a unit of data increment arrives at the central site. In this chapter, a unit of data increment is assumed to be a complete XML document.

We consider an online data integration system which contains a mediator and a number of wrappers (see Figure 4.1).

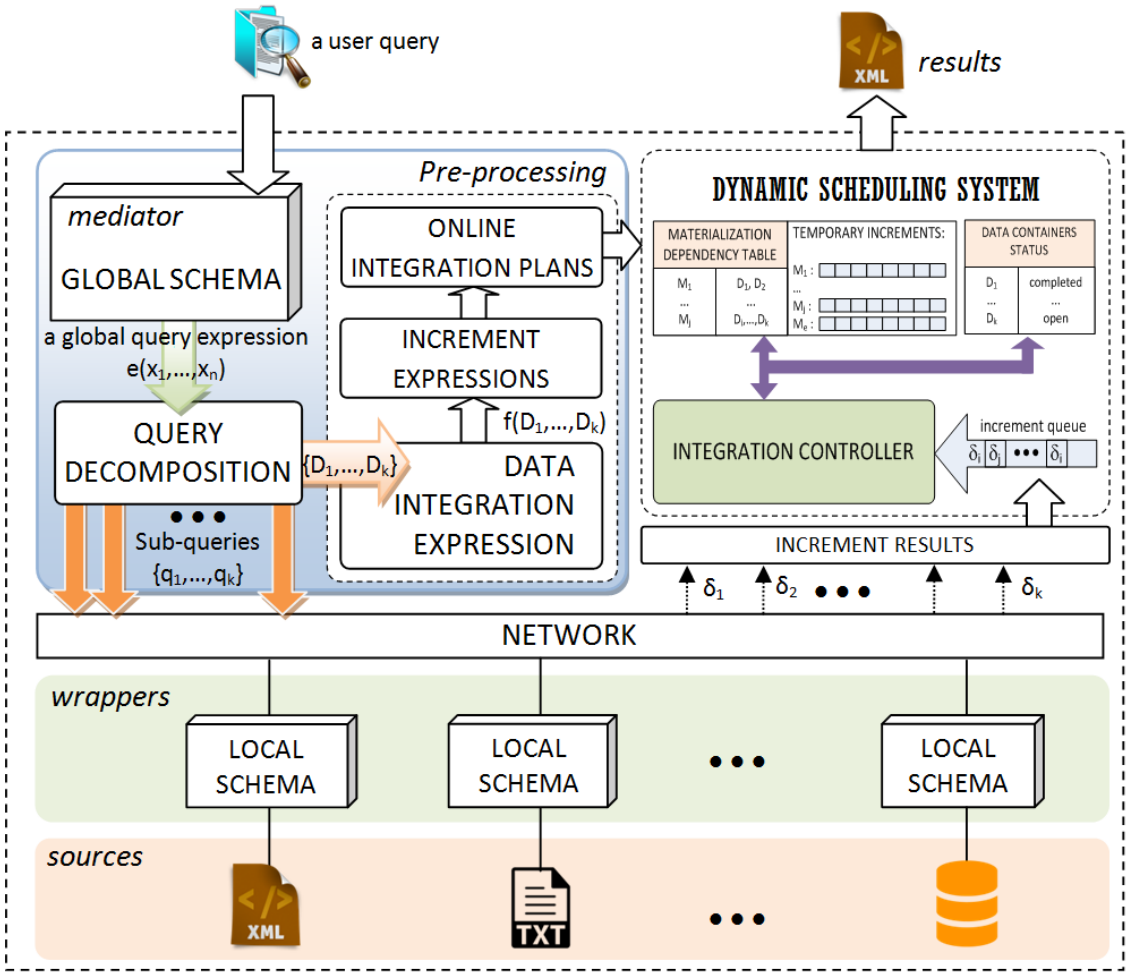

Figure 4.1: Architecture of an online integration system

The architecture of the online data integration system consists of the following parts:

1. Pre-processing.

   The pre-processing module includes a mediator, a global schema, a query decomposition system, and a plan generator. A mediator converts a user request into a global query expression. Then, the query decomposition algorithm decomposes it into a set of sub-queries. In the plan generator, a series of transformations are performed to obtain a set of online integration plans.

2. Query Decomposition.
   Query decomposition is one of the important keys to construct an efficient data integration expression. It employs a mapping of a global view into a set of source schemas, and a strategy to utilize a global view and distributed multi-database environment to obtain an optimal decomposition result.

3. Data Sources.
   In data integration systems, data sources are likely to be heterogeneous and somehow were not designed to support integration. The format of data sources may vary from well-structured data, for example relational databases, to unstructured data like text files. XML has been chosen as a standard for data exchange since it allows encoding of both structured and unstructured data.

4. Wrappers.
   Database systems at highly autonomous remote sites are likely have various data structures. Data integration of the database systems requires a data transformation process into a designated structure. In ths thesis, wrappers at the remote sites are employed to transform the various structures into an XML data model.

5. Dynamic Scheduling System.
   The dynamic scheduling system consists of a sliding window, a dependency table, a data container status list, and an increment queue. It incorporates an algorithm to manage a sequence of increments from the remote sites and provides the best way to process them. Every increment in the sliding window is labeled with a priority label accordingly to their increment type, and then they are processed based on their priorities.

A mediator-wrapper data integration system consists of a global schema G, a set of source schemas S, and a mapping between the source and global schemas $M : G \rightarrow S$. A set of source schemas is defined as $S = \{\langle s_i : x_i \rangle : i = 1, \ldots, n\}$, where $\langle s_i : x_i \rangle$ represents a dataset located at remote site s_i. A global schema is an XML view, and is defined as $G = \{g_1, \ldots, g_n\}$, where each element g_i is a query over S and can be expressed as $g_i = e(\langle s_1 : x_1 \rangle, \ldots, \langle s_n : x_n \rangle)$. A query for g_i is built on XML algebra over S. Meanwhile, a wrapper has the responsibility of transforming data structures at the remote sites into an XML model.

The outline of the online data integration process considered in this thesis is as follows:

1. A user sends a request formulated over a global schema G to the central site of the online data integration system. Since G contains a set of XML views, then the query submitted to the central site is in the form of a high level query language for XML like XQuery.

2. In the first step of query processing at the central site, a *mediator* transforms a user query expressed in XQuery language into a query expressed in XQuery Core through a series of normalization processes [73].

3. Then, based on a mapping $M : G \rightarrow S$, a query in the XQuery Core is translated into a *global query expression* over a set of source schemas S and XML algebraic operations.

4. During this process, a user query can be optimized using the standard techniques of syntax-based optimization, e.g. moving *selection* operations (σ) before binary operations, and *projection* (π) are pushed to the top of a *global query expression*.

5. The computation of a *global query expression* is performed at both the remote sites and the central site depending on their CPU performance and network characteristics. In general, the remote sites have lower resources than the central site. Thus, sending complex queries to the remote sites for computation reduces the performance of an integration system. On the other hand, sending simple queries to the remote sites yields a large amount of results to be transmitted to the central site, and requires a higher communication cost. A better performance is obtained by balancing of the resource consumptions at the central and remote sites with the communication cost.

6. The decomposition process produces a set of sub-queries $\{q_i : i = 1, \ldots, k\}$ and transforms a *global query expression* into an equivalent expression $f(q_1, \ldots, q_k)$ where a set of sub-queries $\{q_i : i = 1, \ldots, k\}$ is sent to the remote sites for processing. The computation results of sub-queries $\{q_i : i = 1, \ldots, k\}$ arrive at the central site in the form of data containers $\{D_i(\mathcal{G}) : i = 1, \ldots, k\}$.

   To simplify a notation, we use D_i instead of a full notation of a data container which includes its schema $D_i(\mathcal{G})$. A data container D_i is the computation result of sub-query q_i.

7. An online integration system allows us to compute a unit of increment without requiring all data to be available at the central site. In this chapter, we

consider a unit of increment as a complete XML document. We transform a *data integration expression* into a set of *increment expression*s to include processing of a data increment δ_i against data containers and the current results. For every data container D_i, an increment expression g_i is generated.

8. In the next step, we generate an *online integration plan* d_i for every *increment expression* constructed earlier. d_i is a set of steps to compute an *increment expression*, where every step is an XML algebraic operation over a data increment (δ_i) and a data container D_j or a materialization M_a.

9. In the execution phase, every data increment δ_i at a data container D_i is integrated into the current result of computing by execution of an *online integration plan* d_i prepared earlier. Finally, the central site sends the final result of computation to the user.

In this thesis, a *projection* operation is always performed at the end after integration processing is completed.
In the following sections, every part of the online data integration system is described.

4.2 Global Query Expression

Since user queries submitted to the central site use a global schema G, they must be transformed into a *global query expression* according to schema mapping $M : G \rightarrow S$, to allow computation on a set of source schemas S at the remote sites.

Definition 28. *Let* $\{\langle s_i : x_i\rangle : i = 1, \ldots, n\}$ *be a set of datasets located at the remote sites. A* global query expression $e(\langle s_1 : x_1\rangle, \ldots, \langle s_n : x_n\rangle)$ *is an expression built from the XML algebra operations of selection, projection, join, antijoin, union, and remote datasets.*

Example 4.1. *A global query expression*
Let $g_1 = e(\langle s_1 : x_1\rangle), g_2 = e(\langle s_1 : x_2\rangle \sim \langle s_1 : x_3\rangle), g_3 = e(\langle s_1 : x_4\rangle \sim \langle s_1 : x_5\rangle), g_4 = e(\langle s_1 : x_6\rangle), g_5 = e(\langle s_2 : x_7\rangle), g_6 = e(\langle s_3 : x_8\rangle \bowtie \langle s_3 : x_9\rangle)$ be XML views in a global schema G. Let a user query: "Get all complete book documents including subject description from `book` data (g_1) where their editors are not book writers (g_2) and their subjects are not in the top 10 lists (g_3). In addition, we ask for book and author data from g_4 which are written by any `author`, but does not exist the same book and same author in g_6".

The user query above can be written as a query in XQuery language as follows:

```
(for $x in doc("g1.xml")//book,
     $y in doc("g2.xml")//editor,
     $z in doc("g3.xml")//subject
 where $x//ed_email=$y/email and $x/subject=$z
 return $x)
 union
(for $b in
    (for $u in doc("g4.xml")//book,
         $v in doc("g5.xml")//author
     where $u//aut_email=$v/email
     return <result>{$u,$v}</result>)
 where empty(doc("g6.xml")//book[title=$b//book/title])
 return $b)
```

In the next step, the user query in the XQuery language is translated into a query in the XQuery Core language. The query is broken into two sub-queries, namely the parts before and after the *union* operator, to simplify the translation process, . The sub-queries is translated into queries in XQuery Core. Then, a *union* operation on the sub-queries is perfomed. The first sub-query is as follows:

```
for $x in doc("g1.xml")//book,
    $y in doc("g2.xml")//editor,
    $z in doc("g3.xml")//subject
where $x//ed_email=$y/email and $x/subject=$z
return $x
```

It is translated into a query in XQuery Core as follows:

```
for $x in doc(g1.xml) return
   for $y in doc(g2.xml) return
      for $z in doc(g3.xml) return
         if (not(empty(
            for $aut_email in $x/authors/aut_email return
               for $email in $y/email return
                  if (eq($aut_email = $email)) then
                    for $book_sbj in $x/subject return
                       for $subj in $z return
                          if(eq($book_sbj=$subj)) then $subj else ()
                  else ()
         ))) then
            element result {$x, $y}
         else ()
```

It can be expressed in XML algebra as follows:

$$sub - query\ 1 = (g_1 \bowtie_{\varphi_1} g_2) \bowtie_{\varphi_1} g_3) \tag{4.1}$$

where φ_1 = "aut_email=email" and φ_2 = "email=email".
The second sub-query is as follows:

```
(for $b in
  (for $u in doc("g4.xml")//book,
       $v in doc("g5.xml")//author
   where $u//aut_email=$v/email
   return <result>{$u,$v}</result>)
     where empty(doc("g6.xml")//book[title=$b//book/title])
     return $b)
```

It is translated into a query in XQuery Core as follows:

```
for $b in (
   for $u in doc("g4.xml") return
      for $v in doc("g5.xml") return
         if (not(empty(
            for $aut_email in $u/authors/aut_email return
               for $email in $v/email return
                  if(eq($aut_email=$email)) then $email else ()
         ))) then
            element result {$u, $v}
         else ()
   ) return
   for $c in doc("g6.xml") return
      if (empty(
         for $aut_email in $b/authors/aut_email return
            for $email in $c/email return
               if (eq($aut_email = $email)) then $email else ()
         )) then
            element result {$x}
         else ()
```

This sub-query can be expressed in an XML algebra expression as:

$$sub - query\ 2 = ((g_4 \bowtie_{\varphi_3} g_5) \sim_{\varphi_4} g_6) \tag{4.2}$$

where φ_3 = "aut_email=email" and φ_4 = "aut_email=email".
The complete query in XQuery Core is obtained by a *union* operation over Equations 4.1 and 4.2, and can be expressed as follows:

$$user\ query = ((g_1 \bowtie_{\varphi_1} g_2) \bowtie_{\varphi_2} g_3) \cup ((g_4 \bowtie_{\varphi_3} g_5) \sim_{\varphi_4} g_6) \tag{4.3}$$

To simplify notations, we remove the condition expressions in the following sections without changing the meaning of the operations.

After the user query is translated into an XML algebra expression, the mediator transforms it into a *global query expression* based on the schema mappings provided. Figure 4.2 shows a *global query expression* tree which is written as follows:

$$e = ((\langle s_1 : x_1\rangle \bowtie (\langle s_1 : x_2\rangle \sim \langle s_1 : x_3\rangle)) \bowtie (\langle s_1 : x_4\rangle \sim \langle s_1 : x_5\rangle)) \cup$$
$$((\langle s_1 : x_6\rangle \bowtie \langle s_2 : x_7\rangle) \sim (\langle s_3 : x_8\rangle \bowtie \langle s_3 : x_9\rangle))$$

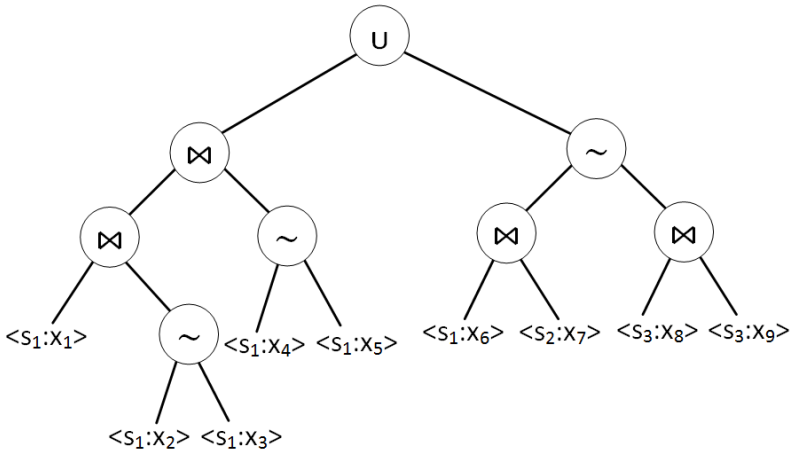

Figure 4.2: A global query expression tree for the user query in Example 4.1

The *global query expression* example is intentionally left as a non-optimized query for showing the purpose of the online integration process. □

4.3 Decomposition of a Global Query Expression

The computation of a *global query expression* can be performed by sending simple requests to the remote sites to retrieve XML documents from all datasets ($\langle s_i : x_i\rangle$). Then, the integration process is performed at the central site.

Example 4.2. *Processing of a global query expression*
Let $e = ((\langle s_1 : x_1\rangle \bowtie (\langle s_1 : x_2\rangle \sim \langle s_1 : x_3\rangle)) \bowtie (\langle s_1 : x_4\rangle \sim \langle s_1 : x_5\rangle)) \cup ((\langle s_1 : x_6\rangle \bowtie \langle s_2 : x_7\rangle) \sim (\langle s_3 : x_8\rangle \bowtie \langle s_3 : x_9\rangle))$ be a *global query expression*. Processing of an expression e is performed by sending simple queries to obtain individual datasets from the remote sites, as shown in the following:

1. Get a data container x_1 from remote site s_1
2. Get a data container x_2 from remote site s_1
3. Get a data container x_3 from remote site s_1

4. Get a data container x_4 from remote site s_1
5. Get a data container x_5 from remote site s_1
6. Get a data container x_6 from remote site s_1
7. Get a data container x_7 from remote site s_2
8. Get a data container x_8 from remote site s_3
9. Get a data container x_9 from remote site s_3

□

Query decomposition by sending simple requests to the remote sites as in Example 4.2 may have a low performance in the following situations:

1. Datasets x_1, x_2 and x_3 at the remote site s_1 contain a large number of XML documents to be transferred to the central site s_1, and
2. Processing of $\langle s_1 : x_2\rangle \sim \langle s_1 : x_3\rangle$ at the central site results in an empty set data container.
3. Therefore, the computation of a sub-query $((\langle s_1 : x_1\rangle \bowtie (\langle s_1 : x_2\rangle \sim \langle s_1 : x_3\rangle)) \bowtie (\langle s_1 : x_4\rangle \sim \langle s_1 : x_5\rangle))$ returns an empty set of data containers as a result.

In the above situation, the processing of an expression e requires transmission costs to send datasets $\langle s_1 : x_1\rangle, \langle s_1 : x_2\rangle, \langle s_1 : x_3\rangle, \langle s_1 : x_4\rangle, \langle s_1 : x_5\rangle$ to the central site. The costs can be reduced if the all computations are performed at the remote site s_1.

The *global query expression* is decomposed into a number of sub-expressions such that an optimal query processing is obtained. The query decomposition process is initiated by the central site, and employs the remote sites to perform part of the computations and send the results back to the central site for further computation.

Definition 29. *Let* $\{\langle s_i : x_i\rangle : i = 1, \ldots, n\}$ *be a set of XML data containers located at the remote sites.* Query decomposition *is a process, that transforms a global query expression* $e(\langle s_1 : x_1\rangle, \ldots, \langle s_n : x_n\rangle)$ *into an equivalent expression* $f(q_1, \ldots, q_k)$. $\{q_i : i = 1, \ldots, k\}$ *is a set of query expressions where* $q_i = e_i(\langle s_i : x_{i_1}\rangle, \ldots, \langle s_i : x_{i_j}\rangle)$, *and* $\{\langle s_i : x_{i_1}\rangle, \ldots, \langle s_i : x_{i_j}\rangle\} \subseteq \{\langle s_1 : x_1\rangle, \ldots, \langle s_n : x_n\rangle\}$. *More than one query expression might be sent to the same remote site. The results*

of processing expression $f(q_1, \ldots, q_k)$ *are identical with the results of processing a global query expression* $e(\langle s_1 : x_1\rangle, \ldots, \langle s_n : x_n\rangle)$.

Query decomposition is one of the key components in a data integration system. Therefore, a good query decomposition strategy is required to gain optimal results. The strategy must consider that the processing over a distributed multi-database is determined by the remote sites parameters, including network characteristics, CPU performance and query complexity.

A very simple decomposition strategy allows the remote sites to do most of the computations if they have good resources. In this strategy, the central site scans the largest possible sub-queries of a global query expression tree where all of their terminal nodes (data containers) are located at the same remote site.

Example 4.3. *A query decomposition strategy*
Let $e(\langle s_1 : x_1\rangle, \ldots, \langle s_3 : x_9\rangle)$ be a global query expression in Example 4.1. A decomposition strategy to send the largest sub-queries to the remote sites is shown in Figure 4.3.

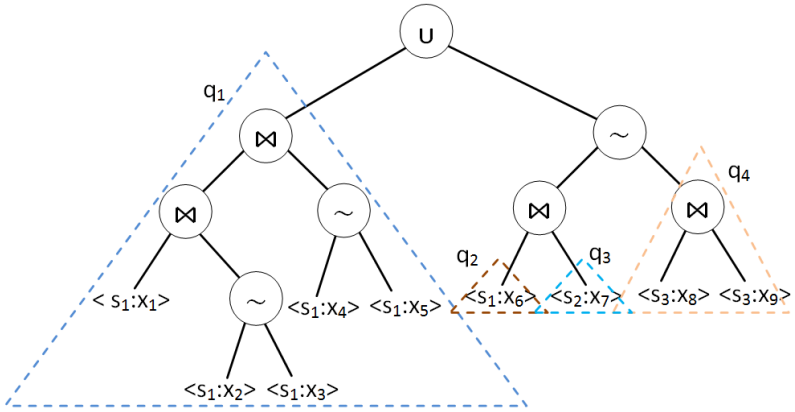

Figure 4.3: A decomposition strategy to send the largest sub-queries to the remote sites

Then, decomposition of a global query expression $e(\langle s_1 : x_1\rangle, \ldots, \langle s_3 : x_9\rangle)$ produces a set of sub-queries as follows:

$$
\begin{aligned}
q_1 &= e(\langle s_1 : x_1\rangle, \langle s_1 : x_2\rangle, \langle s_1 : x_3\rangle, \langle s_1 : x_4\rangle, \langle s_1 : x_5\rangle) \\
&= ((\langle s_1 : x_1\rangle \bowtie (\langle s_1 : x_2\rangle \sim \langle s_1 : x_3\rangle)) \bowtie (\langle s_1 : x_4\rangle \sim \langle s_1 : x_5\rangle)) \\
q_2 &= e(\langle s_1 : x_6\rangle) = \langle s_1 : x_6\rangle \\
q_3 &= e(\langle s_2 : x_7\rangle) = \langle s_2 : x_7\rangle \\
q_4 &= (\langle s_3 : x_8\rangle, \langle s_3 : x_9\rangle) = (\langle s_3 : x_8\rangle \bowtie \langle s_3 : x_9\rangle).
\end{aligned}
$$

A global query expression $e(\langle s_1 : x_1\rangle, \ldots, \langle s_3 : x_9\rangle)$ is transformed into an equivalent expression $f(q_1, q_2, q_3, q_4) = q_1 \cup ((q_2 \bowtie q_3) \sim q_4)$. Computation of an expression $f(q_1, q_2, q_3, q_4)$ is performed by sending sub-queries q_1, q_2 to the remote site s_1, sub-query q_3 to the remote site s_2 and q_4 to the remote site s_3. Then, the computation results are sent back to the central site for integration. □

A good decomposition strategy considers all the available resources including the remote sites, the central site and network characteristics. It balances processing between the central and remote sites in order to improve the performance of processing a query. A decomposition process is performed in three steps. First, we transform a *global query expression* into several large sub-expressions such that all of their arguments are located at one remote site. Thus, a sub-expression can be entirely processed at a single remote site.

In the next step, we find a set of smaller sub-expressions which have lower total costs than the ones generated in the earlier process. The total costs include cost for processing at the remote sites, cost to transmit the results, and cost for processing at the central site. In this step, the algorithm searches for an optimal balance between the amounts of processing at the remote and the central sites.

An optimal *query decomposition* is a strategy that finds a set of sub-expressions $q_1, \ldots, q_k$ whose processing at the remote sites and later on processing of data integration expression $f(D_1, \ldots, D_k)$ at the central site requires the lowest cost. In this thesis, the total cost of query processing is formulated as:

$$TCost(\alpha) = PCost + CCost \tag{4.4}$$

$TCost(\alpha)$ represents the total cost of processing sub-expression α, $PCost$ is the aggregation of individual operation costs in a sub-expression, and $CCost$ represents the communication costs required to transmit the results to the central site.

For an individual operator, the processing cost is determined by the IO cost to access data in a persistent storage, and the CPU cost to execute the operation. Since the IO cost is significantly larger than the CPU cost, then the CPU cost can be typically ignored. At a remote site, The $PCost$ depends on how actual documents are read from a physical semistructured database, how many disk blocks are accessed and the characteristics of secondary storage; at the central site, it is mainly determined by the IO costs to read and write temporary results.

On the other hand, the communication cost ($CCost$) becomes an important factor as the remote sites do not have uniform communication characteristics. The communication characteristic is mainly determined by the distance between the

central site and the remote sites, the physical communication media, the size of document to be transfered, and other aspects related to the network. In spite of that, an assumption is made that the communication cost is mainly determined by the size of documents sent to the central site, since it is a predictable component of the cost and which can be used for an optimization strategy.

The largest sub-expressions are discovered by a systematic labeling of the operation nodes in a syntax tree of a *global query expression* with the identifiers of the remote sites as its arguments. Labeling starts from the operations just above a leaf level in the syntax tree and continues towards the root node. Systematic labeling can be performed by a post-order traversal of the tree starting from the root node. It is performed by the following steps:

1. The label of the current node is checked. If the "site" label is not empty, then we return its label and the process is finished, which means that the whole subtree is located at the same remote site.

2. When step no 1 is not satisfied, the left subtree is traversed and its site label (s_i) is obtained. These complete steps are performed recursively.

3. After step 2 is done, the right subtree is visited and its remote site information=s_j is obtained. These complete steps are performed recursively.

4. If $s_i = s_j$ then set the "site" label of the current node with value s_i. If $s_i \neq s_j$ then set the label of current node with an empty character (space). In the case of a node of a unary operator, set the label of current node=s_i.

At the end of the labeling process, a set of the largest sub-expressions whose child nodes are all labeled with the same "site" label is obtained.

Example 4.4. *Identification of the largest sub-expressions in a global query expression*

Let $e(\langle s_1 : x_1\rangle, \ldots, \langle s_3 : x_9\rangle)$ be a global query expression in Example 4.1. The largest sub-expressions for the global query expression are shown in Figure 4.4, and the sub-expression is as follows:

$$
\begin{aligned}
q_1 &= e(\langle s_1 : x_1\rangle, \langle s_1 : x_2\rangle, \langle s_1 : x_3\rangle, \langle s_1 : x_4\rangle, \langle s_1 : x_5\rangle) \\
&= ((\langle s_1 : x_1\rangle \bowtie (\langle s_1 : x_2\rangle \sim \langle s_1 : x_3\rangle)) \bowtie (\langle s_1 : x_4\rangle \sim \langle s_1 : x_5\rangle)) \\
q_2 &= e(\langle s_1 : x_6\rangle) = \langle s_1 : x_6\rangle \\
q_3 &= e(\langle s_1 : x_7\rangle) = \langle s_1 : x_7\rangle \\
q_4 &= e(\langle s_1 : x_8\rangle, \langle s_1 : x_9\rangle) = (\langle s_1 : x_8\rangle \bowtie \langle s_1 : x_9\rangle).
\end{aligned}
$$

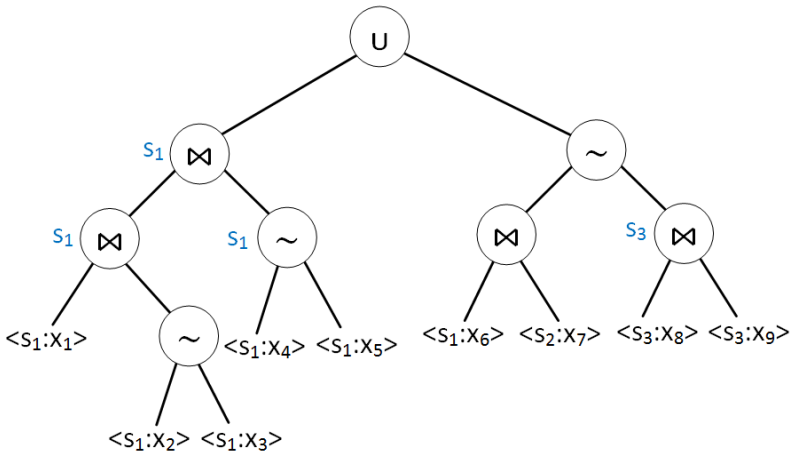

Figure 4.4: Identification of sub-expressions in a global query expression tree

□

Next, we find an optimal processing strategy separately for every sub-expression. Let $\alpha(p, q)$ be a sub-expression with an operation α and the sub-expressions p and q such that all data containers processed by p and q are located in the same remote site. Let $PCost_c(\alpha)$ be the processing cost of sub-expression α at the central site, and $PCost_r(\alpha)$ be the processing cost of sub-expression α at the remote site.

Then, there exist five possible strategies of processing $\alpha(p, q)$ and its corresponding cost:

1. Both sub-expressions p and q, and operation α are processed at a remote site and the results of α are transmitted to the central site. In this situation, we remove communication cost to transfer the result of processing sub-expression p and q from their total costs. Then, the total cost to compute sub-expression $\alpha(p, q)$ is:

$$\begin{aligned} TCost_{(1)}(\alpha) =& TCost_r(\alpha) + TCost(p) + TCost(q) - CCost(p) - CCost(q) \\ =& PCost_r(\alpha) + CCost(\alpha) + TCost(p) + TCost(q) - CCost(p) - \\ & CCost(q) \end{aligned}$$

2. Both sub-expressions (p, q) are processed at remote sites, and the results are transmitted to the central site. An operation α is processed at the central site. Then, the total costs is:

$$TCost_{(2)}(\alpha) = PCost_c(\alpha) + TCost(p) + TCost(q)$$

3. Sub-expression p is processed at a remote site, meanwhile q and an operation

α are processed at the central site. The total processing costs is:

$$TCost_{(3)}(\alpha) = PCost_c(\alpha) + TCost(p) + TCost(q) - CCost(q)$$

4. Sub-expression q is processed at the remote site, meanwhile p and an operation α is processed at a central site. The total processing costs is:

$$TCost_{(4)}(\alpha) = PCost_c(\alpha) + TCost(p) - CCost(p) + TCost(q).$$

5. Both sub-expressions p and q are processed at the central site as well as operation α. Then, the total processing costs is:

$$TCost_{(5)}(\alpha) = PCost_c(\alpha) + TCost(p) + TCost(q) - CCost(p) - CCost(q)$$

The best query decomposition is obtained by a comparison of total costs of the alternatives listed above, and the variant with the lowest processing costs is selected for processing of sub-expression $\alpha(p, q)$.

All possible strategies is obtained by three following steps:

1. First, the nodes in a sub-expression are labeled with the positive natural numbers such that if a node is labeled with n then its child nodes are labeled with $(n * 2)$ and $(n * 2 + 1)$.

2. Next, all possible labelings of the operation nodes in the sub-expression with either "remote" or "central" is identified, such that if a node is labeled with "remote" then all its child nodes are labeled with "remote" as well.

   The labeling of the nodes with numbers done in the previous steps allows for a quick selection of the child nodes. Algorithm 1 demonstrates a sample implementation of this process. A *boolean* variable *hasSub* is used to identify whether a node contains sub-expression(s) or not. A node with a label n has at least a sub-expression if there exists any node which is labeled $n*2$ or/and $n * 2 + 1$ in the current labeling. A function `IndexOf`(*string* `label`,*char* `-`), is employed to obtain the left most dash character '`-`' in the `label`. The character '`-`' represents the next node that has not been processed.

3. Finally, the labels generated in the previous steps are used to calculate the total cost for each variant.

   Algorithm 2 is an implementation example for step 3 above.

Algorithm 1 Generate all combinations

Require: Sub-expression in array of node (N[max])
Ensure: A set of sub-expressions

```
 1: function GenerateAll(N[max])
 2:     strSeed={'-',...,'-'}; j=0; Remote='R'; Central='C'; strResult[max];
 3:     if (strSeed[1]='-') then
 4:         stack.push(SetData(N,strSeed, Central, 1));
 5:         stack.push(SetData(N,strSeed, Remote, 1));
 6:     end if
 7:     while (!stack.empty()) do
 8:         strCombination=stack.pop();
 9:         thisPos = IndexOf(strCombination, '-');
10:         if (thisPos>0) then
11:             stack.push(SetData(N,strCombination, Central, thisPos));
12:             stack.push(SetData(N,strCombination, Remote, thisPos));
13:         else
14:             strResult[j++]=strCombination;
15:         end if
16:     end while
17:     return strResult;
18: end function

19: function IndexOf(str, ch)
20:     for (i=1 to str.length) do
21:         if (str[i]=ch) then
22:             return i
23:         end if
24:         return 0
25:     end for
26: end function

27: function SetData(N[max], str, value, pos)
28:     Remote='R'; Central='C'; hasSub=false;
29:     if (pos≥ max) then
30:         return str;
31:     end if
32:     if (str[pos]=='-') then
33:         str[pos] = value;
34:     end if
35:     if (N[pos*2] or N[pos*2+1] exists in N) then
36:         hasSub=true;
37:     else
38:         str[pos] = "_";
39:     end if
40:     if !(value=Central or (value=Remote and !hasSub)) then
41:         str = SetData(N,str, Remote, pos*2);
42:         str = SetData(N,str, Remote, pos*2 + 1);
43:     end if
44:     return str;
45: end function
```

Algorithm 2 Find a combination with the minimum total cost

Require: A set of combinations Result[r]
Ensure: The best combination

```
1: function FindBest(Result[x])
2:     BestIndex=0; s=""; Central='C'; Remote='R'; BestCost=0;
3:     for (x=0 to r-1) do
4:         s=Result[x];
5:         TotalCost=TCost(s,1,2,3);
6:         if (TotalCost<BestCost) then
7:             BestCost=TotalCost;
8:             BestIndex=x;
9:         end if
10:    end for
11:    return Result[BestIndex];
12: end function

13: function TCost(s, α, p, q)
14:    TLeft = 0; TRight = 0;
15:    if (s[p]<>"_") then
16:        TLeft=TCost(s,p,p*2,p*2+1);
17:    end if
18:    if (s[q]<>"_") then
19:        TRight=TCost(s,q,q*2,q*2+1);
20:    end if
21:    if (s[a]="R") then
22:        Total = PCost_r(α) + CCost(α) + TLeft − CCost(p) + TRight −
       CCost(q);
23:    else
24:        Total = PCost_c(α) + TLeft + TRight;
25:    end if
26:    return Total
27: end function
```

Example 4.5. *Decomposition strategy*
We consider a global query expression $e = ((\langle s_1 : x_1\rangle \bowtie (\langle s_1 : x_2\rangle \sim \langle s_1 : x_3\rangle)) \bowtie (\langle s_1 : x_4\rangle \sim \langle s_1 : x_5\rangle)) \cup ((\langle s_1 : x_6\rangle \bowtie \langle s_2 : x_7\rangle) \sim (\langle s_3 : x_8\rangle \bowtie \langle s_3 : x_9\rangle))$, and its syntax tree as shown in Figure 4.5(a). It identifies that all arguments in the sub-expression (α) are from a single remote site s_1. For sub-expression $\alpha(\langle s_1 : x_1\rangle, \ldots, \langle s_1 : x_5\rangle)$ a syntax tree is constructed as shown in Figure 4.5(b), such that all nodes are labeled with a positive integer and the root node is labeled with the number "1".

In Algorithm 1 we construct an array of characters to represent all possible decomposition labelings of a sub-expression. To simplify the decomposition process, we employ an array of characters to store every node's label. The array starts from

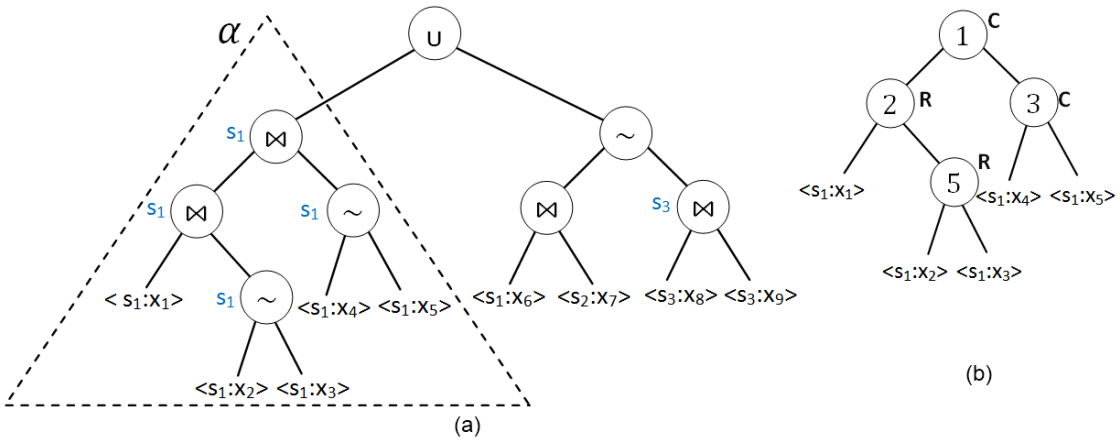

Figure 4.5: (a) A syntax tree of a *global query expression* (b) An example of decomposition strategy to balance query processing between a remote and the central site.

index 1 which is associated with the root node. The array of labels {R,R,R,_,R} represents that all nodes are processed at a remote site. Index 4 of the array is an underscore character (_) which represents that the left child of node number 2 does not exist in the syntax tree. The result of these algorithms is a set of labeling arrays: {R,R,R,_,R}, {C,R,R,_,R}, {C,R,C,_,R}, {C,C,R,_,R}, {C,C,R,_,C}, {C,C,C,_,R}, and {C,C,C,_,C}.

In the next step, the strategies constructed from the previous steps are evaluated to find the best decomposition strategy. THe notation of x_i is used to replace $\langle s_i : x_i \rangle$ for simplification of notation. For a strategy {C,R,C,_,R} as in Figure 4.5(b), the central site sends sub-expression $\beta(x_1, x_2, x_3) = \beta(x_1, \theta(x_2, x_3))$ to the remote site s_1 for computation. The central site receives the results of computing from remote site s_1 and store them in a data container.

The total cost of sub-expression $\alpha(\beta(x_1, \theta(x_2, x_3)), \gamma(x_4, x_5))$ with the strategy labeled as {C,R,C,_,R} is as follows:

$$TCost(\alpha) = PCost_c(\alpha) + TCost(\beta) + TCost(\gamma) - CCost(\gamma)$$

where x_1, x_2, x_3, x_4, and x_5 represent data containers at the remote site s_1, and

$$\begin{aligned}
TCost(\beta) =& PCost_r(\beta) + CCost(\beta) + TCost(x_1) + TCost(\theta) - CCost(x_1) - \\
& CCost(\theta) \\
TCost(\gamma) =& PCost_c(\gamma) + TCost(x_4) + TCost(x_5) \\
TCost(\theta) =& PCost_r(\theta) + CCost(\theta) + TCost(x_2) + TCost(x_3) \\
TCost(x_1) =& PCost_r(x_1) + CCost(x_1)
\end{aligned}$$

$$TCost(x_2) = PCost_r(x_2) + CCost(x_2)$$
$$TCost(x_3) = PCost_r(x_3) + CCost(x_3)$$
$$TCost(x_4) = PCost_r(x_4) + CCost(x_4)$$
$$TCost(x_5) = PCost_r(x_5) + CCost(x_5)$$

Then, the total cost for processing sub-expression q_1 is:

$$\begin{aligned} TCost(\alpha) = & PCost_c(\alpha) + PCost_r(\beta) + PCost_r(\theta) + PCost_c(\gamma) + PCost_r(x_1) + \\ & PCost_r(x_2) + PCost_r(x_3) + PCost_r(x_4) + PCost_r(x_5) + CCost(\beta) + \\ & CCost(x_4) + CCost(x_5) \end{aligned}$$

The total cost to retrieve documents from remote datasets x_1, x_2, x_3, x_4, x_5 includes the CPU cost for *selection* operations, and communication costs to bring the documents to the central site. The costs of *selection* operations can be ignored, because their transmission costs are typically larger than the CPU cost. Therefore, the total cost of sub-expression $q_1(\alpha)$ can be formulated as:

$$\begin{aligned} TCost(\alpha) = & PCost_c(\alpha) + PCost_r(\beta) + PCost_r(\theta) + PCost_c(\gamma) + CCost(\beta) + \\ & CCost(x_4) + CCost(x_5) \end{aligned}$$

The same decomposition algorithm is applied to sub-expression $\alpha(x_8, x_9)$.

After the decomposition process, we obtain an equivalent expression $f(q_1, q_2, q_3, q_4, q_5, q_6)$, where q_1, q_2, q_3, q_4, and q_6 are sub-expressions as follows:

$$q_1 = e(x_1, x_2, x_3) = x_1 \bowtie (x_2 \sim x_3)$$
$$q_2 = e(x_4) = \sigma(x_4)$$
$$q_3 = e(x_5) = \sigma(x_5)$$
$$q_4 = e(x_6) = \sigma(x_6)$$
$$q_5 = e(x_7) = \sigma(x_7)$$
$$q_6 = e(x_8, x_9) = x_8 \bowtie x_9$$

□

When the CPU cost is typically ignored, the cost of processing a sub-expression $a(r, s)$ at a remote site ($PCost_r(a)$) is determined by IO costs to read datasets from a secondary storage at the remote sites r and s, and the algorithm used to implement physical algebraic operations. The processing cost for a *join* operation

$(r \bowtie s)$ is:

$$PCost_r(a) = |n_r| * |n_s|$$

where $|n_r|$ and $|n_s|$ are the estimated number of data in r and s respectively. In a more complex situation, IO costs include the number of blocks accessed, the estimated average size of data, and the size of the remote sites' main memory.

Computation results from the remote sites are received at the central site in the form of data containers. They are expected to fit in the main memory of the central site. In some cases, computation results are materialized at the central site in order to avoid re-computation and to reduce computation cost. Operations on materialized data require IO costs to read data from the secondary storage and to write the computation results back to the secondary storage, if materialized updates are required.

The communication cost ($CCost$) is determined by the amount of data transferred to the central site. In Example 4.5, $CCost(x_4)$ and $CCost(x_5)$ represent communication costs to send documents from x_4 and x_5 at the remote site s_1 to the central site. Meanwhile, $CCost(\beta)$ is the cost to transfer results of computing sub-expression β to the central site. If $|n|$ is an estimated amount of result documents and s is an estimated average size of a document to be transferred, then $CCost(\beta) = |n| * s$.

At the end of the decomposition process we obtain an equivalent query expression $f(q_1, ..., q_k)$ as a result, where $f(q_1, ..., q_k) \equiv e(\langle s_1 : x_1\rangle, \ldots, \langle s_n : x_n\rangle)$, and $q_i = e_i(x_1, \ldots, x_n)$.

4.4 Data Integration Expression

The decomposition of a global query expression produces a query expression $f(q_1, ..., q_k)$, where sub-queries $q_i : i = 1, \ldots, k$ are sent to the remote sites for computation. Then, the result of computations are transfered to the central site in the form of data containers, and a *data integration expression* is generated.

Definition 30. *Let $f(q_1, ..., q_k)$ be an equivalent expression as a result of decomposition of an expression $e(\langle s_1 : x_1\rangle, \ldots, \langle s_n : x_n\rangle)$. Let D_i be a data container to collect the result of processing q_i at a remote site. A data integration expression $f(D_1, ..., D_k)$ is an expression obtained from $f(q_1, \ldots, q_k)$ by a systematic replacement of the symbols $q_1, \ldots, q_k$ with the names of data containers $D_1, \ldots, D_k$.*

Example 4.6. *A data integration expression*
Let $f(q_1, q_2, q_3, q_4, q_5, q_6)$ be a query expression as a result of decomposition in Example 4.5, as shown in Figure 4.6 (a).

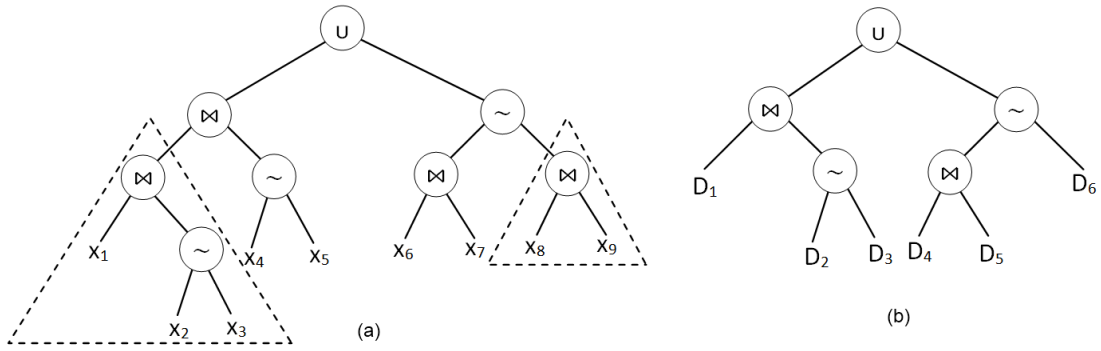

Figure 4.6: (a) A syntax tree of a *global query expression* and decomposition strategy to balance central and remote site processing (b) A data integration expression

We send sub-queries q_1, q_2, q_3 and q_4 to remote site s_1 for computation, a sub-expression q_5 to remote site s_2 and q_6 to remote site s_3. The computation results of all sub-queries from the remote sites are transferred to the central site, which collects all data in a set of data containers $\{D_i : i = 1, 2, \ldots k\}$ for integration. Then, a data integration expression $f(D_1, \ldots, D_6)$ is generated. $D_1 - D_6$ represent data containers at the central site to collect results from sub-queries $q_1 - q_6$ respectively, such that:

$$f(D_1, \ldots, D_6) = (D_1 \bowtie (D_2 \sim D_3)) \cup ((D_4 \bowtie D_5) \sim D_6) \tag{4.5}$$

.
Figure 4.6(b) shows a syntax tree of the *data integration expression.*

□

During a decomposition process, it may happen that some sub-expressions are identical. This leads to a situation where a data container D_i shows up more than once in a data integration expression. The duplicate sub-expressions can be treated as a single data container to reduce the number of data which have to be transfered to the central site.

In the next sub section, we make an assumption that a data container D_i is unique and exists in a single argument in a data integration expression in order to simplify the idea of creating an *increment expression.* Meanwhile, a discussion on duplicate sub-expressions will be covered in the processing of concurrent increments section.

4.4.1 Increment Expression

In the next step, a data integration expression is transformed into a form which allows us to compute it step by step when a data increment occurs at a data container at the central site. Let δ_{i_j} be an increment of a data container D_i for $j = 1, \ldots, n$. Then, a data container is formed as $D_i = \delta_{i_1} \cup \delta_{i_2} \cup \ldots \cup \delta_{i_n}$. In this thesis a data increment is presented by a *union* operation, while in some other models it may have different properties or operations. In this section, we consider a data increment unit (δ_{ij}) is a complete XML document, and leave increment of fragmented documents for the next discussion.

Definition 31. *Let $\{D_i : i = 1, \ldots, k\}$ be a set of data containers in a data integration expression $f(D_1, \ldots, D_k)$. A materialization is denoted as M_a, and is defined as computation result of a data integration expression $M_a = h_a(D_1, \ldots, D_k) : a = 1, \ldots, j$. A materialization is usually stored in a secondary storage. In a special case, M_a can be an identity function of a data container D_i.*

Example 4.7. *Materializations in a data integration expression*
Let $f(D_1, \ldots, D_6)$ be a data integration expression as in Figure 4.6 (b). In on-line data integration, materializations might be needed to increase its performance without re-computing a particular sub-expression. Four materializations are constructed in $f(D_1, \ldots, D_6)$ as follows:

$$M_1 = (D_2 \sim D_3)$$
$$M_2 = (D_1 \bowtie (D_2 \sim D_3)) = (D_1 \bowtie M_1)$$
$$M_3 = (D_4 \bowtie D_5)$$
$$M_4 = ((D_4 \bowtie D_5) \sim D_6) = (M_3 \sim D_6)$$

A data integration expression tree which includes materializations is shown in Figure 4.7. □

The intermediate materializations are stored in a mapping table and will be used to identify the dependencies of an intermediate materialization to any data container later on.

Definition 32. *Let D_i be a data container, δ_i be an increment of D_i, and $\{M_1, \ldots, M_j\}$ be a set of intermediate materializations. An increment expression is denoted as $g_i(\delta_i, M_1, \ldots, M_j)$, and is defined as an expression to compute an increment data (δ_i) against intermediate materializations. g_i has a form of left/right deep*

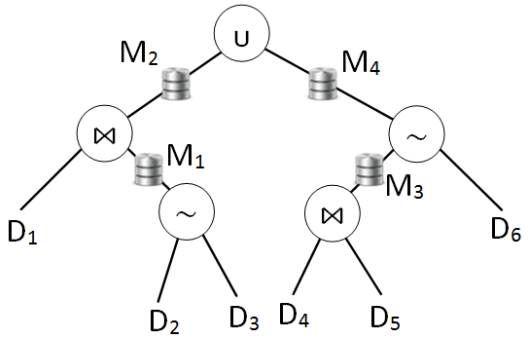

Figure 4.7: A data integration expression includes materializations

expression such that $g_i = \omega_j(\ldots(\omega_2(\omega_1(\delta_i, M_1), M_2), \ldots), M_j)$. $\omega_n : n = 1 \ldots j$ *operates on XML operators: join, antijoin, and union.*

Example 4.8. *A mapping table of intermediate materializations*

Data Container	Dependent materializations
D_1	M_1
D_2	M_1, M_2
D_3	M_2
D_4	M_3, M_4
D_5	M_3, M_4
D_6	M_4

□

Theorem 2. Any data integration expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ can be always transformed into one of the following equivalent expressions:

$$f(D_1, \ldots, D_i, \ldots, D_k) \cup g_i(\delta_i, M_1, \ldots, M_j) \tag{4.6}$$

$$f(D_1, \ldots, D_i, \ldots, D_k) \sim g_i(\delta_i, M_1, \ldots, M_j) \tag{4.7}$$

where $g_i(\delta_i, M_1, \ldots, M_j)$ is an increment expression.

Proof. Theorem 2:

For $k = 1$, an expression f operates on a unary operator. Then, according to the proof for a *selection* operation (see rule 2), we show that $f(D_1 \cup \delta_1) = f(D_1) \cup g(\delta_1)$ is true. g and f are identical functions.

For $k = 2$, an expression f is operated on binary operators. Then, according to the proofs for *union, join,* and *antijoin* operations (see rule 3-12), we show that $f(D_1, D_2 \cup \delta_2) = f(D_1, D_2) \cup g(\delta_2, D_1)$ and $f(D_1, D_2 \cup \delta_2) = f(D_1, D_2) \sim g(\delta_2, D_1)$ are true;

Assume that an expression with k data containers $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ can be transformed into either an expression in Equation 4.6 or 4.7.

Let $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_{k+1})$ be an extension of $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ with a data container D_{k+1} by operation of *union* ($\cup$), *join* ($\bowtie$), or *antijoin* ($\sim$). Transformation to all possible extensions is as follows:

1. We extend an expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ with a *union* operation ($\cup$) where $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is the first argument and D_{k+1} is the second argument. The expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is transformed as follows:

$$
\begin{aligned}
&= f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) \cup D_{k+1} \\
&= (f(D_1, \ldots, D_k) \cup g(\delta_i, M_1, \ldots, M_j)) \cup D_{k+1}, \text{we apply rule 3} \\
&= (f(D_1, \ldots, D_k) \cup D_{k+1}) \cup (g(\delta_i, M_1, \ldots, M_j) \cup D_{k+1})) \\
&= f(D_1, \ldots, D_{k+1}) \cup g(\delta_i, M_1, \ldots, M_j, D_{k+1}). \\
&\quad \text{Since } M_a = h_a(D_1, \ldots, D_k) : a = 1, \ldots, j, \text{ we are able to form} \\
&\quad M'_a = h'_a(D_1, \ldots, D_{k+1}) : a = 1, \ldots, k. \text{ Then:} \\
&= f(D_1, \ldots, D_{k+1}) \cup g(\delta_i, M'_1, \ldots, M'_k)
\end{aligned} \tag{4.8}
$$

2. We extend an expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ with a *union* ($\cup$) operator where D_{k+1} is the first argument and $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is the second argument. The expression f is transformed as follows:

$$
\begin{aligned}
&= D_{k+1} \cup f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) \\
&= D_{k+1} \cup (f(D_1, \ldots, D_k) \cup g(\delta_i, M_1, \ldots, M_j)), \text{we apply rule 3} \\
&= (D_{k+1} \cup f(D_1, \ldots, D_k)) \cup (D_{k+1} \cup g(\delta_i, M_1, \ldots, M_j)) \\
&= f(D_1, \ldots, D_{k+1}) \cup g(\delta_i, M_1, \ldots, M_j, D_{k+1}), \\
&\quad \text{Since } M_a = h_a(D_1, \ldots, D_k) : a = 1, \ldots, j, \text{ we are able to form} \\
&\quad M'_a = h'_a(D_1, \ldots, D_{k+1}) : a = 1, \ldots, k. \text{ Then:} \\
&= f(D_1, \ldots, D_{k+1}) \cup g(\delta_i, M'_1, \ldots, M'_k)
\end{aligned} \tag{4.9}
$$

3. We extend an expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ with a *join* ($\bowtie$) operation where $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is the first argument and D_{k+1} is the second argument. The expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is transformed

as follows:

$$
\begin{aligned}
&= f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) \bowtie D_{k+1} \\
&= (f(D_1, \ldots, D_k) \cup g(\delta_i, M_1, \ldots, M_j)) \bowtie D_{k+1}, \text{we apply rule 4} \\
&= (f(D_1, \ldots, D_k) \bowtie D_{k+1}) \cup (g(\delta_i, M_1, \ldots, M_j) \bowtie D_{k+1}) \\
&= f(D_1, \ldots, D_{k+1}) \cup g(\delta_i, M_1, \ldots, M_j, D_{k+1}) \\
&\quad \text{Since } M_a = h_a(D_1, \ldots, D_k) : a = 1, \ldots, j, \text{ we are able to form} \\
&\quad M'_a = h'_a(D_1, \ldots, D_{k+1}) : a = 1, \ldots, k. \text{ Then:} \\
&= f(D_1, \ldots, D_{k+1}) \cup g(\delta_i, M'_1, \ldots, M'_k) \qquad (4.10)
\end{aligned}
$$

4. We extend an expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ with a *join* ($\bowtie$) operator where D_{k+1} is the first argument and $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is the second argument. The expression f is transformed as follows:

$$
\begin{aligned}
&= D_{k+1} \bowtie f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) \\
&= D_{k+1} \bowtie (f(D_1, \ldots, D_k) \cup g(\delta_i, M_1, \ldots, M_j)), \text{we apply rule 4} \\
&= (D_{k+1} \bowtie f(D_1, \ldots, D_k)) \cup (D_{k+1} \bowtie g(\delta_i, M_1, \ldots, M_j)) \\
&= f(D_1, \ldots, D_{k+1}) \cup g(\delta_i, M_1, \ldots, M_j, D_{k+1}). \\
&\quad \text{Since } M_a = h_a(D_1, \ldots, D_k) : a = 1, \ldots, j, \text{ we are able to form} \\
&\quad M'_a = h'_a(D_1, \ldots, D_{k+1}) : a = 1, \ldots, k. \text{ Then:} \\
&= f(D_1, \ldots, D_{k+1}) \cup g(\delta_i, M'_1, \ldots, M'_k) \qquad (4.11)
\end{aligned}
$$

5. We extend an expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ with an *antijoin* ($\sim$) operator where $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is the first argument and D_{k+1} is the second argument. The expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is transformed as follows:

$$
\begin{aligned}
&= f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) \sim D_{k+1} \\
&= (f(D_1, \ldots, D_k) \cup g(\delta_i, M_1, \ldots, M_j)) \sim D_{k+1}, \text{we apply rule 5:} \\
&= (f(D_1, \ldots, D_k) \sim D_{k+1}) \cup (g(\delta_i, M_1, \ldots, M_j) \sim D_{k+1}) \\
&= f(D_1, \ldots, D_{k+1}) \cup g(\delta_i, M_1, \ldots, M_j, D_{k+1}) \\
&\quad \text{Since } M_a = h_a(D_1, \ldots, D_k) : a = 1, \ldots, j, \text{ we are able to form} \\
&\quad M'_a = h'_a(D_1, \ldots, D_{k+1}) : a = 1, \ldots, k. \text{ Then:} \\
&= f(D_1, \ldots, D_{k+1}) \cup g(\delta_i, M'_1, \ldots, M'_k) \qquad (4.12)
\end{aligned}
$$

6. We extend an expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ with an *antijoin* ($\sim$) operator where D_{k+1} is the first argument and $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is the second argument. The expression f is transformed as follows:

$$
\begin{aligned}
&= D_{k+1} \sim f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) \\
&= D_{k+1} \sim (f(D_1, \ldots, D_k) \cup g(\delta_i, M_1, \ldots, M_j)), \text{we apply rule 6:} \\
&= (D_{k+1} \sim f(D_1, \ldots, D_k)) \sim g(\delta_i, M_1, \ldots, M_j) \\
&= f(D_1, \ldots, D_{k+1}) \sim g(\delta_i, M_1, \ldots, M_j) \qquad (4.13)
\end{aligned}
$$

An extension of an increment expression with an operation either *join*($\bowtie$), *antijoin*($\sim$), or *selection*(σ) results an expression in the form of either Equation in 4.6 or 4.7. Therefore, this proves that these rules are complete for *join, antijoin* and *union* operations. □

Transformation of a data integration expression into an increment expression is by application of distributivity properties, and is performed by the following steps:

1. We replace D_i in a data integration expression with $(D_i \cup \delta_i)$.

2. We use XML algebra rules (1-12) explained in section 3.5 to move δ_i inside a data integration expression such that it forms an expression either in Equation 4.6 or Equation 4.7.

Example 4.9. *Transformation of a data integration expression $f(D_1, \ldots, D_k)$ into increment expressions.*
Let $D_1, \ldots, D_6$ be data containers in an expression $f(D_1, \ldots, D_6) = (D_1 \bowtie (D_2 \sim D_3)) \cup ((D_4 \bowtie D_5) \sim D_6)$ (see Equation 4.5). Let δ_1 be an increment of data container D_1, and $g_1(\delta_i, M_1, \ldots, M_j)$ be an increment expression of D_1. Transformation of $f(D_1, \ldots, D_6)$ into g_1 is performed step by step as follows:

$$
\begin{aligned}
&= (D_1 \bowtie (D_2 \sim D_3)) \cup ((D_4 \bowtie D_5) \sim D_6), \textit{replace } D_1 \textit{ with } (D_1 \cup \delta_1) \\
&= ((D_1 \cup \delta_1) \bowtie (D_2 \sim D_3)) \cup ((D_4 \bowtie D_5) \sim D_6), \textit{ apply rule } (4) \\
&= ((D_1 \bowtie (D_2 \sim D_3)) \cup (\delta_1 \bowtie (D_2 \sim D_3))) \cup ((D_4 \bowtie D_5) \sim D_6), \textit{ apply rule } (3) \\
&= ((D_1 \bowtie (D_2 \sim D_3)) \cup ((D_4 \bowtie D_5) \sim D_6)) \cup (\delta_1 \bowtie (D_2 \sim D_3)) \\
&= f(D_1, \ldots, D_6) \cup (\delta_1 \bowtie (D_2 \sim D_3)) \\
&= f(D_1, \ldots, D_6) \cup (\delta_1 \bowtie M_1)
\end{aligned}
$$

Then, transformation of $f(D_1, \ldots, D_6)$ with an increment at D_1 produces the result:

$$\delta_1 : f(D_1, \ldots, D_6) \cup (\delta_1 \bowtie M_1)$$

where $g_1 = (\delta_1 \bowtie (D_2 \sim D_3)) = (\delta_1 \bowtie M_1)$ as an increment expression of D_1.

An increment expression for a data increment at a data container D_2 is obtained by transformation of a data integration expression step by step as follows:

$$\begin{aligned}
&= (D_1 \bowtie (D_2 \sim D_3)) \cup ((D_4 \bowtie D_5) \sim D_6), \textit{replace } D_2 \textit{ with } (D_2 \cup \delta_2) \\
&= (D_1 \bowtie ((D_2 \cup \delta_2) \sim D_3)) \cup ((D_4 \bowtie D_5) \sim D_6), \textit{ apply rule } (5) \\
&= (D_1 \bowtie ((D_2 \sim D_3) \cup (\delta_2 \sim D_3))) \cup ((D_4 \bowtie D_5) \sim D_6), \textit{ apply rule } (8) \\
&= ((D_1 \bowtie (D_2 \sim D_3)) \cup (D_1 \bowtie (\delta_2 \sim D_3))) \cup ((D_4 \bowtie D_5) \sim D_6), \textit{ apply rule } (3) \\
&= f(D_1, \ldots, D_6) \cup (D_1 \bowtie (\delta_2 \sim D_3))
\end{aligned}$$

Then, an increment expression $g_2 = (D_1 \bowtie (\delta_2 \sim D_3))$ is obtained.

Following the same transformation procedures above, we obtain the increment expressions for the remaining data containers. After all transformations are completed, we obtain the following expressions:

$$\begin{aligned}
&\delta_1 : f(D_1, \ldots, D_6) \cup (\delta_1 \bowtie M_1); \\
&\delta_2 : f(D_1, \ldots, D_6) \cup (D_1 \bowtie (\delta_2 \sim D_3)); \\
&\delta_3 : f(D_1, \ldots, D_6) \sim (\delta_3 \sim ((D_4 \bowtie D_5) \sim D_6)) = f(D_1, \ldots, D_6) \sim (\delta_3 \sim M_4); \\
&\delta_4 : f(D_1, \ldots, D_6) \cup ((\delta_4 \bowtie D_5) \sim D_6); \\
&\delta_5 : f(D_1, \ldots, D_6) \cup ((D_4 \bowtie \delta_5) \sim D_6); \\
&\delta_6 : f(D_1, \ldots, D_6) \sim (\delta_6 \sim M_1)).
\end{aligned}$$

Therefore, we obtain increment expressions as follows:

$$g_1 = (\delta_1 \bowtie M_1); \tag{4.14}$$

$$g_2 = (D_1 \bowtie (\delta_2 \sim D_3)); \tag{4.15}$$

$$g_3 = (\delta_3 \sim M_4); \tag{4.16}$$

$$g_4 = ((\delta_4 \bowtie D_5) \sim D_6); \tag{4.17}$$

$$g_5 = ((D_4 \bowtie \delta_5) \sim D_6); \tag{4.18}$$

$$g_6 = (\delta_6 \sim M_1) \tag{4.19}$$

Figure 4.8 shows the syntax tree of expressions for online processing of data increments $\delta_1, \delta_2, \delta_3, \delta_4, \delta_5$ and δ_6. The results of an increment expression g_i are combined with the previous final materialization $M_e = f(D_1, D_2, D_3, D_4, D_5, D_6)$ to produce a new state of processing.

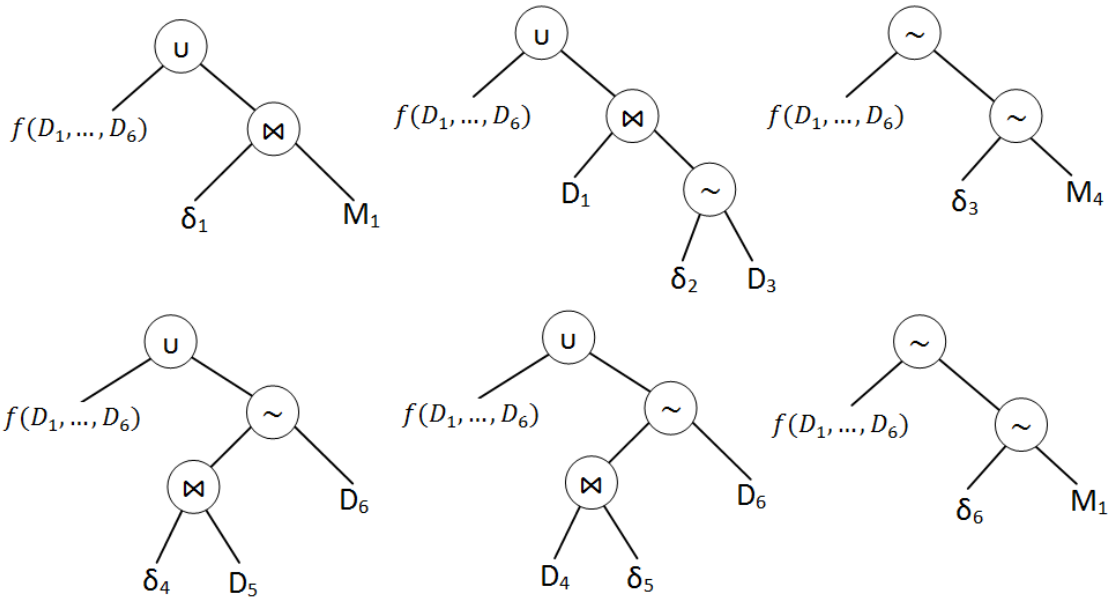

Figure 4.8: Increment expressions for the data integration expression in Figure 4.6 (b)

□

At the end of the transformation process we obtain a set of increment expressions $\{g_1, \ldots, g_k\}$ and a list of intermediate materializations $\{M_1, \ldots, M_j\}$ to compute the increment data. Since an intermediate materialization is a result of computations of a data integration expression $M_i = h_a(D_1, \ldots, D_k)$, an intermediate materialization update is performed in the same way as processing of the data increment.

4.4.2 Online Integration Plans

In the next step, we prepare an online integration plan for every increment expression generated at the earlier stage.

Definition 33. *Let D_i be a data container, and δ_i be an increment of D_i. Let $g_i(\delta_i, M_1, \ldots, M_j)$ be an increment expression for δ_i where $M_i : i = 1 \ldots j$ are materializations. An online integration plan for D_i is denoted as d_i, and is defined as $d_i : \langle p_1; \ldots; p_m \rangle$. $p_j : j = 1, \ldots, m$ is a sequence of steps where in each*

step a simple XML algebra operation is evaluated. The computation results of the operations are transferred from step to step until all steps are computed.

The transformation of an increment expression g_i into an online integration plan d_i is performed by the following procedure:

1. Since $g_i(\delta_i, M_1, \ldots, M_j) = \omega_j(\ldots(\omega_2(\omega_1(\delta_i, M_1), M_2)\ldots)M_j)$, every expression $\omega_i : i = 1, \ldots, j$ is mapped into a corresponding step from the inner-most to the outer most XML algebraic operation.

2. Then, a step to update a final materialization (M_e) with the last result of the increment computation is appended ($M'_e = M_e \cup g_i$ or $M'_e = M_e \sim g_i$).

3. In the next step, a data container D_i is updated with $(D_i \cup \delta_i)$.

4. We identify all intermediate materializations $M_a = h_a(D_1, \ldots, D_k)$ which are affected by the data increment. The identification process utilizes a lookup table for materialization dependencies generated earlier.

5. Lastly, we perform the same procedures to compute every materialization $M_a : a = 1, \ldots, j$ identified, but without a step to update the data container.

Example 4.10. *The transformation of an increment expression into an online integration plan.*
Let g_2 be an increment expression generated in Example 4.9. Transformation of an increment expression $g_2 = (D_1 \bowtie (\delta_2 \sim D_3))$ into an online integration plan d_2 is performed as follows:

1. In the first step, we map an expression $(\delta_2 \sim D_3)$ into a step $\Delta_1 = (\delta_2 \sim D_3)$ and an expression $(D_1 \bowtie \Delta_1)$ into a step $\Delta_2 = (D_1 \bowtie \Delta_1)$;

2. Then, we append $M_e = (M_e \sim \Delta_2)$ to combine the computation result of previous step with previous final materialization;

3. Next, we create a step to update the data container D_2: $D_2 = (D_2 \cup \delta_2)$;

4. An intermediate materialization M_1 is identified from the dependency table to be affected by the update. M_1 is a computation result of a data integration expression $h_1(D_2, D_3) = (D_2 \sim D_3)$, therefore $h_1(D_1, D_2)$ is transformed into an increment expression $g_{M1} = (\delta_2 \sim D_3)$, and a plan to update M_1 is generated as follows: $d_{M1} : \Delta_{M1} = (\delta_2 \sim D_3); M_1 = (M_1 \cup \Delta_{M1})$. These processing steps are appended to the steps produced earlier.

Then, an online integration plan for increment expression g_2 is as follows:

$$p_1 : \Delta_1 = (\delta_2 \sim D_3);$$
$$p_2 : \Delta_2 = D_1 \bowtie \Delta_1;$$
$$p_3 : M_e = (M_e \sim \Delta_3);$$
$$p_4 : D_2 = (D_2 \cup \delta_2);$$
$$p_5 : \Delta_{M1} = (\delta_2 \sim D_3);$$
$$p_6 : M_1 = (M_1 \cup \Delta_{M1}).$$

Since $\Delta_{M1} = \Delta_1$, we remove computation of $\Delta_{M1} = (\delta_2 \sim D_3)$ such that:

$$d_2 : \Delta_1 = (\delta_2 \sim D_3); \Delta_2 = D_1 \bowtie \Delta_1; M_e = (M_e \sim \Delta_2); D_2 = (D_2 \cup \delta_2);$$
$$M_1 = (M_1 \cup \Delta_1).$$

The same procedures are applied to obtain all online integration plans for the data integration expression in Equation 4.5. Then, the following online integration plans are generated:

$$d_1 : \Delta_1 = (\delta_1 \bowtie M_1); M_e = (M_e \cup \Delta_1); D_1 = (D_1 \cup \delta_1); M_2 = (M_2 \cup \Delta_1). \quad (4.20)$$
$$d_2 : \Delta_1 = (\delta_2 \sim D_3); \Delta_2 = (D_1 \bowtie \Delta_1); M_e = (M_e \sim \Delta_2); D_2 = (D_2 \cup \delta_2);$$
$$M_1 = (M_1 \cup \Delta_1). \quad (4.21)$$
$$d_3 : \Delta_1 = (\delta_3 \sim M_4); M_e = (M_e \sim \Delta_1); D_3 = (D_3 \cup \delta_3); M_1 = (M_1 \sim \delta_3);$$
$$M_2 = (M_2 \sim (D_1 \sim \delta_3)). \quad (4.22)$$
$$d_4 : D_4 = (D_4 \cup \delta_4); \Delta_1 = (\delta_4 \bowtie D_5); \Delta_2 = (\Delta_1 \sim D_6); M_e = (M_e \cup \Delta_2);$$
$$D_4 = (D_4 \cup \delta_4); M_3 = (M_3 \cup \Delta_1); M_4 = (M_4 \cup \Delta_2). \quad (4.23)$$
$$d_5 : \Delta_1 = (D_4 \bowtie \delta_5); \Delta_2 = (\Delta_1 \sim D_6); M_e = (M_e \cup \Delta_2); D_5 = (D_5 \cup \delta_5);$$
$$M_3 = (M_3 \cup \Delta_1); M_4 = (M_4 \cup \Delta_2). \quad (4.24)$$
$$d_6 : \Delta_1 = (\delta_6 \sim M_1); M_e = (M_e \sim \Delta_1); D_6 = (D_6 \cup \delta_6);$$
$$M_4 = (M_4 \sim \delta_6). \quad (4.25)$$

□

In some cases, we can find an equivalent increment expression $g_i'(\delta_i, M_1', \ldots, M_j')$ from $g_i = (\delta_i, M_1, \ldots, M_j)$ by application of distributive, associative, and commutative laws of the XML algebra operators. Both increment expressions g_i and g_i' produce the same results of computation. The equivalent increment expression

allows us to generate an alternative plan and find the best plan by an evaluation of a cost model for each plan. An algorithm to obtain the best equivalent expression is not covered in this thesis.

Example 4.11. We consider an increment expression $g_x = (((\delta_1 \bowtie D_2) \bowtie D_3) \bowtie M_3)$. An online integration plan for increment expression g_x is as follows:

$$d_x : \Delta_1 = (\delta_1 \bowtie D_2); \Delta_2 = (\Delta_1 \bowtie D_3); \Delta_3 = (\Delta_2 \bowtie M_3);$$
$$M_e = (M_e \cup \Delta_3); D_1 = (D_1 \cup \delta_1); M_1 = (M_1 \cup \Delta_1).$$

By an application of associativity rule, an increment expression $g_x = (((\delta_1 \bowtie D_2) \bowtie D_3) \bowtie M_3)$ is equivalent to $g'_x = (\delta_1 \bowtie ((D_2 \bowtie D_3) \bowtie M_3))$. The expression can be written as $g'_x = (\delta_1 \bowtie M_4)$, where $M_4 = ((D_2 \bowtie D_3) \bowtie M_3)$. Furthermore, the transformation of $(\delta_1 \bowtie M_4)$ creates an online integration plan as follows:

$$d_x{}' : \Delta'_1 = (\delta_1 \bowtie M_4); M_e = (M_e \cup \Delta'_1); D_1 = (D_1 \cup \delta_1); \Delta'_2 = \delta_1 \bowtie D_2;$$
$$M_1 = (M_1 \cup \Delta'_2).$$

Although an increment expression $g'_x = (\delta_1 \bowtie M_4)$ seems to be simpler than $g_x = (((\delta_1 \bowtie D_2) \bowtie D_3) \bowtie M_3)$, its integration plan $d_x{}'$ might not be better than d_x because it introduces an intermediate materialization $M_4 = ((D_2 \bowtie D_3) \bowtie M_3)$. An intermediate materialization M_4 needs to be updated whenever increment data occurs at data container D_2, D_3, D_4 and D_5, while an intermediate materialization M_3 is updated when increments from data containers D_4 and D_5 occur. □

4.5 Scheduling of Online Integration Plans

In the compilation process, a mediator prepares a set of online integration plans in which every plan is assigned to a data container $D_i : i = 1, \ldots, k$. They include at least a step to compute a relatively small increment against a large data container or a materialization, which is a step to update either data container, an intermediate materialization, or a final materialization. An operation on materialized data requires an IO cost to load them into the main memory and store them back to the persistent storage.

At run time, a sequence of data increments can be processed accordingly to their incoming order. However, an optimization of the online integration system might be obtained if we modify the sequence of data increments such that IO cost is reduced.

4.5.1 Static Scheduling

In an online data integration system, a data increment δ_i received at data container D_i triggers execution of an online integration plan d_i prepared earlier. The static scheduling mode executes data increments in the order in which they arrive at the central site. A static scheduling strategy is performed in the following manner:

Algorithm 3 Static scheduling algorithm

```
1: prepare an online integration plan d_i for every data container D_i
2: while increment data occurs do
3:     pick an online integration plan for corresponding increment data prepared earlier
4:     for each step in integration plan d_i do
5:         load data to the main memory if needed
6:         perform an algebraic operation
7:         if (step=materialization update) then
8:             perform writing to a secondary storage
9:         end if
10:    end for
11: end while
```

Example 4.12. *Static scheduling for a sequence of data increments*
Let δ_i be an increment at a data container D_i in the data integration expression of Equation 4.5. Let d_i be an online integration plan for δ_i as in Example 4.10. We consider data increments arrive at the central site in the following sequence: $\delta_{1a} \leftarrow \delta_{1b} \leftarrow \delta_2 \leftarrow \delta_{3a} \leftarrow \delta_4 \leftarrow \delta_{5a} \leftarrow \delta_{3b} \leftarrow \delta_{5b}$, where δ_{1a} and δ_{1b} represent two different increments at D_1. A static scheduling strategy processes the sequence of data increments according to the following steps:

1. We prepare online integration plans for all data containers.
2. Then, we process the first data increment δ_{1a} and compute it by the execution of all steps in online integration plan d_1. Processing of d_1 is performed as follows:
    a. Load an increment data δ_1 into main memory.
    b. Load an intermediate materialization M_1 into the main memory.
    c. Perform an operation $(\delta_1 \bowtie M_1)$ and store the result in main memory (Δ_1).
    d. Load a final materialization M_e into main memory;
    e. Perfom an operation of $(M_e \cup \Delta_1)$ and write the result back to the secondary storage.

f. Load a data container D_1 into main memory.

g. Perform an operation $(D_1 \cup \delta_1)$ and update to D_1 in main memory.

h. Load an intermediate materialization M_2 into main memory.

i. Compute an operation $(M_2 \cup \Delta_1)$ and write the result back to the secondary storage .

3. Processing the next data increment δ_{1b} requires the execution of the same online integration plan d_1 with the same steps above.

4. The processing is continued with the execution of an online integration plan d_2 for an increment data δ_2, followed by an online integration plan d_3 for a data increment δ_{3a}, d_4 for δ_4, d_5 for δ_{5b}, d_3 for δ_{3b} and finally d_5 for an increment data δ_{5b}.

□

The static scheduling strategy might result in poor performance in the following circumstances:

1. *At the initial stage.* At this stage most of the data containers are empty; therefore executing all the steps in a corresponding integration plan is unlikely to give the correct answer for a user query. Therefore, we can stop further processing steps when a binary operation operates on an increment data against an empty data container. The most important step to be executed in this stage is to update the data container itself.

2. *A sequence of increments at one data container.* The processing of an increment data requires preparation of an online integration plan before the computation can be started, and writing of all materializations after it is completed. Thus, processing two data increments from different data containers in a row requires the preparation of an online integration plan twice. Let a sequence of $\delta_i \leftarrow \delta_j$ be data increments from D_i, D_j respectively. Processing of two data increments from the same data container in a row allows us to reduce the preparation process because they have the same online integration plan. Furthermore, the step to update intermediate materializations is unnecessary, because they are not used in the next computation, and hence this reduces the IO cost. Increment results to be appended into intermediate materialization are stored in a temporary list, and used for materialization update when necessary.

3. *A sequence of small increments.* In some cases, the preparation of processing an increment data is more expensive than processing the increment data itself. In this case, we wait until a certain amount of increments are available before computation is started. This reduces the pre-preparation cost of processing data increments.

4. *At the end stage.* At this stage most data containers are complete. Then, updating intermediate materialization can be ignored if we identify that they are not used in any further computations.

4.5.2 Dynamic Scheduling

The dynamic scheduling algorithm proposed in this thesis eliminates inefficiency of static scheduling described earlier. A monitoring system (see Figure 4.9) is employed to continuously collect the behavior of data increments, data containers and materializations. The monitoring system consists of *an increment queue, an integration controller, a materialization dependency table, temporary increment lists,* and *a data container status table.*

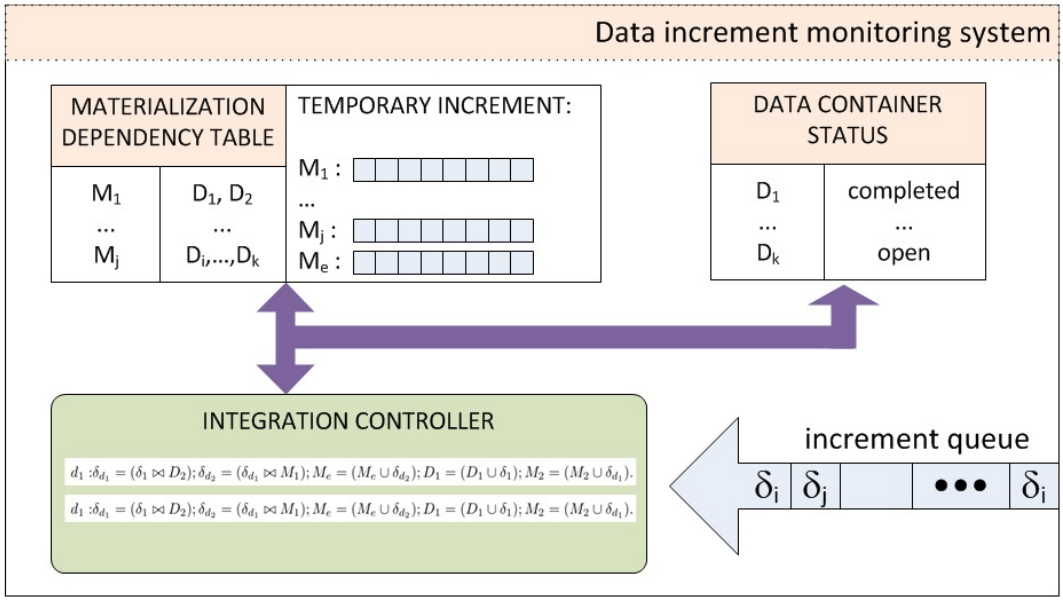

Figure 4.9: A monitoring system for dynamic scheduling

An *increment queue* takes responsibility for managing documents received at the central site, and has a role to serialize concurrent data increments. A *materialization dependency table* maintains the relationship between intermediate materializations and data containers. The table contains information about which materializations are affected by the increment of a data container, and it determines which data containers includes a particular intermediate materialization in

its integration plan. The table is created during the process to generate increment expressions and integration plans.

An *integration controller* contains a collection of plans for all data containers. As the center of a dynamic monitoring system, it utilizes all components to decide whether to continue the current plan, skip some steps of the plan, or terminate it. *Temporary increment lists* are used to retain unprocessed results which later on will be combined with a materialization. A temporary increment list is associated with a materialization, and will be flushed whenever needed. Meanwhile, a *data container status table* is used to determine the data containers' status. For a complete data container, the system identifies which materializations can be excluded from the update process.

Any dynamic scheduling algorithm which employs parallel processing such as pipelining, parallel processors, or multi threading is not covered in this thesis, but can be extended by considering dependencies between steps in online integration plans.

4.5.3 Priority Labeling

The first optimization technique to consider is to minimize the number of documents involved in a single operation. In an online data integration system, an increment expression can be *union*-ed ($\cup$) or *antijoin*-ed ($\sim$) with the previous final materialization (see Equation 4.6 and 4.7). Processing of increment data which has an increment expression in the second form ($\sim$) might potentially reduce the number of documents to be processed. Therefore, by giving a higher priority to the increment expressions in the second form ($\sim$), the performance of the online integration system might be increased.

Example 4.13. *Increment data with the highest priority*
We consider increment expressions which are generated in Example 4.9. D_2, D_3, and D_6 are data containers which have increment expressions of the second form (Equation 4.7). Meanwhile, the other data containers have increment expressions of the first form (Equation 4.6). Therefore, data increments from D_2, D_3, and D_6 are assigned to the highest priorities for execution. □

The next dynamic strategy is to minimize materialization update by the management of increment processing. Let δ_i and δ_j be a sequence of increment data, where δ_i arrives at the central site before δ_j. Two consecutive increments might be in one of the following three possible conditions:

1. Both data increments (δ_i and δ_j) occur at a single data container. For example, δ_{1a} and δ_{1b} are data increments at data container D_1, where $\delta_i = \delta_{1a}$ and δ_j is δ_{1b}. For further discussion, this is named **Type 1**.

2. The data increments (δ_i and δ_j) occur at two different data containers (D_i, D_j), and the data containers form an expression of an intermediate materialization $M_a = h_a(D_i, D_j)$. Processing this sequence of data increments allows us to update the materialization twice with a single process to load from and write to the secondary storate. Hence, it reduces the IO processing cost. As an example, based on the data integration expression in Figure 4.6, δ_i is an increment of data container D_4 (δ_4) and δ_j is an increment of data container D_5 (δ_5). Intermediate materialization M_3 is a computation result of an expression $h_a(D_4, D_5) = (D_4 \bowtie D_5)$. This is an increment in **Type 2**.

3. Both data increments occur at different data containers (D_i, D_j), and the computation of one data increment δ_j requires an updated materialization which involves data container D_i. For an example in Figure 4.6, δ_i is an increment of D_1 (δ_1) and δ_j is increment at data container either D_3, D_4 or D_5. If δ_j is at data container D_3 (δ_3), then an intermediate materialization M_1 must be updated before the computation of increment δ_3 can be started. For further discussion, this is named **Type 3**.

We propose a dynamic scheduling algorithm based on a *sliding window* model. Every increment in a *sliding window* is labeled with a priority number, and then data increments are sorted accordingly to their labels. Priority labeling and sorting are described as follows:

1. A sequence of data increments is obtained from a *sliding window*.

2. Then, the highest priority is given to all increments at data containers which have increment expressions in the second form (see Equation 4.7). If there exist such data increments from more than one data containers, we give the data increments which appear most often a higher priority.

3. If such a data increment in step 2 does not exist, we select a data increment at a data container that appears most often among all data containers in the current *sliding window*;

4. The next data increment is determined by the current one. We choose the next data increment which satisfies Type 1, followed by data increments in

Type 2. The remaining data increments will have the least priorities. Then, the step 3 is repeated until all data increments in the current *sliding window* are scheduled.

Example 4.14. *Priority Labeling*
We consider an integration expression as in Figure 4.6. Data increments arrive at the central site in the following sequence: $\delta_{1a} \leftarrow \delta_{1b} \leftarrow \delta_2 \leftarrow \delta_{3a} \leftarrow \delta_4 \leftarrow \delta_{5a} \leftarrow \delta_{3b} \leftarrow \delta_{5b}$. Suppose, all data increments fit in a *sliding window*. Then, priority labeling and sorting will be performed by the following steps:

1. From a collection of increment expressions prepared from the earlier stage, increment from data container D_3 (δ_{3a} and δ_{3b} are identified having increment expressions in the second form ($\sim$). Therefore, they are set as the first rows of data increments to process, (δ_{3a}, and followed by δ_{3b}).

2. After δ_{3b}, we choose the next data increments which are Type 2. Data container D_2 and D_3 form an expression of materialization $M_1 = (D_2 \sim D_3)$. Therefore, δ_2 is taken as the next increment to be scheduled.

3. The next priority is given to an increment at data container D_1 because there is no increment in Type 1 and 2 after δ_2. Therefore, we choose δ_{1a} as the next increment to process.

4. This is followed by δ_{1b} because they are Type 1.

5. The remaining data increments do not satisfy Type 1 and 2. Then, we consider an increment from D_5 as it appears most often among the rest data increments in the current *sliding window*. δ_{5a} is scheduled;

6. This is followed by δ_{5b} because it occurs at the same data container with the previous increment (Type 1).

7. δ_4 is the last data increment to be scheduled.

Priority labeling produces the following sequence of data increments for processing: $\delta_{3a} \leftarrow \delta_{3b} \leftarrow \delta_2 \leftarrow \delta_{1a} \leftarrow \delta_{1b} \leftarrow \delta_{5a} \leftarrow \delta_{5b} \leftarrow \delta_4$. □

4.5.4 Management of Plans

Dynamic scheduling system proposed in this thesis reduces materialization updates by an *early termination* of plan, and *procrastination* of plan. *Early termination*

eliminates unnecessary computations when the remaining steps in an online integration plan have no impact on the rest of computation. For example, when computation of $(\delta_1 \bowtie D_2)$ in an online integration plan d_1 results in nothing, the rest of the steps of plan d_1 can be ignored, and plan d_1 is ended. Furthermore, when a data container D_2 is empty, the plan d_1 can be terminated earlier.

Meanwhile, a *procrastination of plan* is performed to defer a step to update a materialization when it is not used in the next computation. In a sequence of data increments at a single data container, we collect the result of increment processing in a list and defer the materialization updates. The deferred steps will be executed when the corresponding materialization takes place in the next online integration plan. Algorithm 4 shows a generic dynamic scheduling algorithm.

Algorithm 4 Dynamic scheduling system

```
while (not empty queue) do
    Get increment data from a sliding window;
    Sort increment data based on their priority labels;
    for each increment in (sorted increment data) do
        Prepare online integration plan d_i;
        for each (step p_i in integration plan) do
            if (p_i=materialization (M_a) update then
                if (M_a is not used by next plan) then
                    Store the increments to the designated increment lists;
                    Defer step p_i;
                else
                    if not empty increment lists then
                        Flush the increment lists to M_a
                    end if
                    Execute step p_i;
                end if
            else
                Execute step p_i;
                if (p_i has no result) then
                    Terminate rest steps of current plan d_i;
                end if
            end if
        end for
    end for
    Get to the next sliding window;
end while
```

We introduce increment lists to store computation results which have not been updated to intermediate materializations. Every intermediate materialization has one or two increment lists to keep temporary results of increment expressions. One list is used to store positive increment results ($\cup$) and the other is for negative

increment results ($\sim$).

Definition 34. *Let M_i be an intermediate materialization. Increment lists, which are denoted as $L_i^{\cup}$ and $L_i^{\sim}$, are collections of computation results which have not been updated to M_i. $L_i^{\cup}$ represents the computation results of any increment expression in the first form ($\cup$). $L_i^{\sim}$ collects the computation results of any increment expression in the second form ($\sim$).*

When an intermediate materialization (M_i) update step is deferred, the intermediate computation results (Δ_i) is stored in the designated increment lists. The operation to append the intermediate computation results to the increment lists, is as follows:

1. A positive increment ($\cup$) is appended to the list $L_i^{\cup}$ with the following operation:

$$L_i^{\cup} = L_i^{\cup} \cup (\Delta_i \sim L_i^{\sim})$$

2. A negative increment ($\sim$) is updated to the lists with the following operations:

$$L_i^{\cup} = L_i^{\cup} \sim \Delta_i;$$
$$L_i^{\sim} = L_i^{\sim} \cup \Delta_i$$

Example 4.15. *Plan optimization in dynamic scheduling system*
We consider a data integration expression as in Equation 4.5, and a sequence of data increments arrive at the central site in the following sequence: $\delta_{1a} \leftarrow \delta_{1b} \leftarrow \delta_2 \leftarrow \delta_{3a} \leftarrow \delta_4 \leftarrow \delta_{5a} \leftarrow \delta_{3b} \leftarrow \delta_{5b}$. Let all data containers in the data integration expression be empty before the first data increment is processed. A dynamic scheduling system performs the following steps:

1. Every increment is labeled and sorted by their priorities, and gives us: $\delta_{3a} \leftarrow \delta_{3b} \leftarrow \delta_2 \leftarrow \delta_{1a} \leftarrow \delta_{1b} \leftarrow \delta_{5a} \leftarrow \delta_{5b} \leftarrow \delta_4$.

2. Then, every increment from the sorted and labeled *sliding* windows is computed, starting from increment with the highest priority label.

3. δ_{3a}: The first data increment is δ_{3a} from a data container D_3. After updating D_3, the monitoring system identifies that materialization M_e is empty. Therefore, computation of Δ_1 is not needed, and an *early termination* is

performed. From the execution plan prepared earlier:

$$d_3 : D_3 = (D_3 \cup \delta_3); \Delta_1 = (\delta_3 \sim M_4); M_e = (M_e \sim \Delta_1); M_1 = (M_1 \sim \delta_3); \\ M_2 = (M_2 \sim (D_1 \sim \delta_3)).$$

we execute: $d_3 : D_3 = (D_3 \cup \delta_3)$; and ignore the rest of the steps.

4. δ_{3b}: Next, data increment δ_{3b} at D_3 is processed. Because the final materialization M_e, and intermediate materializations M_1 and M_2 are empty, we execute step $D_3 = (D_3 \cup \delta_3)$; and skip the other steps.

5. δ_2: After δ_{3b}, we compute data increment δ_2 by execution of online integration plan d_2. From the prepared plan below:

$$d_2 : D_2 = (D_2 \cup \delta_2); \Delta_1 = (\delta_2 \sim D_3); \Delta_2 = D_1 \bowtie \Delta_1; M_e = (M_e \sim \Delta_2); \\ M_1 = (M_1 \cup \Delta_1).$$

we execute: $D_2 = (D_2 \cup \delta_2); \Delta_1 = (\delta_2 \sim D_3); M_1 = (M_1 \cup \Delta_1)$.

6. δ_{1a}: All steps in an integration plan d_1 for increment δ_{1a} are executed except a step to update intermediate materialization M_2. Operation to update M_2 is not needed because the next increment comes from the same data container D_1. The increment result Δ_1 is unioned with the increment list $L_2^{\cup}$ because it is in the fist form. Then, from the steps prepared in the online integration plan:

$$d_1 : D_1 = (D_1 \cup \delta_1); \Delta_1 = (\delta_1 \bowtie M_1); M_e = (M_e \cup \Delta_1); M_2 = (M_2 \cup \Delta_1).$$

We execute the following steps:

$$d_1 : D_1 = (D_1 \cup \delta_1); \Delta_1 = (\delta_1 \bowtie M_1); M_e = (M_e \cup \Delta_1); L_2^{\cup} = L_2^{\cup} \cup \Delta_1$$

7. δ_{1b}: To compute a data increment δ_{1b} we perform all prepared steps d_1, and add an operation to update an intermediate materialization M_2 from the increment list $L_2^{\cup}$ as follows:

$$d_1 : D_1 = (D_1 \cup \delta_1); \Delta_1 = (\delta_1 \bowtie M_1); M_e = (M_e \cup \Delta_1); M_2 = (M_2 \sim L_2^{\sim}); \\ M_2 = (M_2 \cup L_2^{\cup}); M_2 = (M_2 \cup \Delta_1).$$

We add an operation to clear the increment lists $L_2^{\cup}$ and $L_2^{\sim}$ after the operation to update an intermediate materialization M_2.

8. δ_{5a}: Since data container D_4 is empty, a step $\Delta_1 = (D_4 \bowtie \delta_5)$ does not need to be executed. Then, from the following plan:

$$d_5 : D_5 = (D_5 \cup \delta_5); \Delta_1 = (D_4 \bowtie \delta_5); \Delta_2 = \Delta_1 \sim D_6; M_e = (M_e \cup \Delta_2); \\ M_3 = (M_3 \cup \Delta_1); M_4 = (M_4 \cup (\Delta_1 \sim D_6)).$$

we execute: $d_5 : D_5 = (D_5 \cup \delta_5)$;

9. δ_{5b}: Processing of data increment δ_{5b} is the same as processing of increment data δ_{5a} because D_4 is empty.

10. δ_4: Processing of data increment δ_4 has to perform all steps in an online integration plan d_4 as follows:

$$d_4 : D_4 = (D_2 \cup \delta_4); \Delta_1 = (\delta_4 \bowtie D_5); \Delta_2 = \Delta_1 \sim D_6; M_e = (M_e \cup \Delta_2); \\ M_3 = (M_3 \cup \Delta_1); M_4 = (M_4 \cup (\Delta_1 \sim D_6)).$$

□

The dynamic scheduling system proposed in this thesis is able to remove unnecessary operations, especially to update materializations which are IO cost expensive. In example 4.15, one online integration plan can be terminated earlier, two materialization updates are deferred, and one plan is terminated.

At the *end stage* of an online data integration system, we consider a *permanent termination* of the plan. *Permanent termination* is a process to cancel all steps in the running plan and stop the current online integration plan when any new increment has no impact on the final result.

Example 4.16. *Permanent termination of an online integration plan* Let data containers D_2, D_3, D_4 and D_5 in Figure 4.6 are complete and computation of $(D_2 \sim D_3)$ and $(D_4 \bowtie D_5)$ return empty results. Then, processing of any data increment at D_1 and D_6 has no impact on the final result. In this case, the dynamic scheduling system forces a *permanent termination* of the current plan, and terminates the integration process. □

Partial results of online integration system are instantly available to the user if the data integration expression has no decremental results (*antijoin* operations).

Otherwise, the correct answers are available to the user after data containers which cause decremental results are complete.

4.5.5 Management of Increment Queue

A data increment which has been processed is removed from the current *sliding window*. Then, the dynamic scheduling system allows the *sliding window* to shift and collect a new increment (see Figure 4.10). The process of labeling and sorting is repeated with a new data increment in a *sliding window*. Sorting of a *sliding window* with an additional data increment is easier because data increments in the previous *sliding window* are sorted.

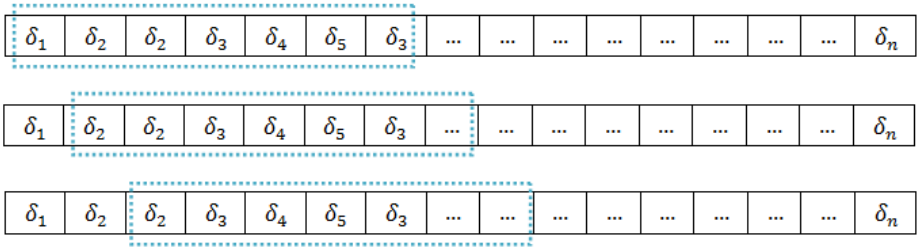

Figure 4.10: Sliding window for processing data increments

The size of the *sliding window* may affect to the performance of the dynamic scheduling system in general. In continuous data increments, the size of a *sliding window* might not be a big problem. The biggest effort is to compute labeling and sorting at the first *sliding window*. Then, for an additional data increment, labeling and sorting a *sliding window* can be performed using a simple algorithm.

For a sequence of data increments which has a significant delay between their arrivals, a big size *sliding window* might be empty before a new data increment arrives. Then, a small size *sliding window* is preferable to the system.

Another approach is to move the *sliding window* after all data increments in a *sliding window* are computed. However, to obtain an optimal processing of a *sliding window* is not covered in this thesis.

A significant delay might exist between arrivals of data increments. The performance of the dynamic scheduling system falls to static scheduling if all steps in a plan are executed before a new data increment arrives. The dynamic scheduling system might employ statistical information of the previous increments to predict the next likely increment to occur. Then, the decision to continue or defer the process to update materialization can be made. In this work it is assumed that the time delay between arrivals of data increments is relatively small.

In the online data integration system described above, processing of a data increment requires updating of materializations to ensure that their values are up-to-date. A dynamic scheduling system minimizes the update of materializations to reduce the IO cost required by the system, but still requires two online integration plan to compute the data increments.

It may happen at the central site, that more than one data increment arrive at about the same time. Processing of those data increments in serial mode would require several operations to update materialization. It is possible to improve the system if processing of several data increments can be performed in a parallel manner. Having a single processor at the central site, processing of parallel data increments can be done by modification of the way to transform a data integration expression into an increment expression.

Chapter 5

Parallel Processing of Data Increments

In the execution phase of a data integration expression, data increments from two or more data containers may arrive at the central site nearly or at the same time. Figure 5.1 illustrates a data integration expression tree which has data increments from two data containers (D_i, D_j) occurring at the same moment. Meanwhile, the online data integration system proposed earlier is designed to process a single increment at a time. Then, according to the algorithms presented earlier, processing the data increments of the data containers D_i and D_j must be performed serially.

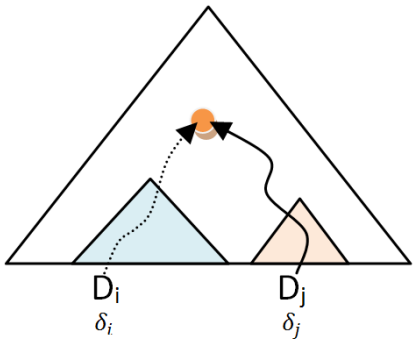

Figure 5.1: Processing of concurrent data increments from two data containers

Processing of two data increments in serial mode requires execution of two online integration plans where each of them includes a step to update a final materialization at the end of the plan. This requires double IO costs to load the final materialization from a secondary storage and double IO costs to write it back to the secondary storage. The system is optimized by modification of the online integration plans to reduce the IO costs by placing the steps to process the final materialization updates such that they can be executed consecutively, one after the other.

A data integration expression for concurrent increments is transformed into one increment expression. Then, an online integration plan to process the concurrent increments is generated for every increment expression. To show how processing of

concurrent data increments works, it is assumed that the central site has a single processor.

In this chapter the online integration system is extended for processing of concurrent data increments. The structure of this chapter is as follows: Section 5.1 describes an example of concurrent data increments, and Section 5.2 discuss the possible approaches to process them. Then, an algorithm to generate an increment expression for concurrent increments is described in Section 5.3. In Section 5.4, a scheduling strategy for the concurrent increments is discussed. Section 5.5 describes the execution of an online integration plan in parallel fashion. The dynamic scheduling system in Section 5.6 extends the scheduling system in Chapter 4 to achieve an efficient processing of concurrent data increments.

5.1 An Example of Data Integration Expression

The data integration expression in Example 5.1 is provided to support the description of processing concurrent data increments.

Example 5.1. *A data integration expression with concurrent increments*
We consider a data integration expression as follows:

$$f(D_1, D_2, D_3, D_4) = (D_1 \bowtie D_2) \sim (D_3 \bowtie D_4) \tag{5.1}$$

The data integration expression is visualized as the following syntax tree:

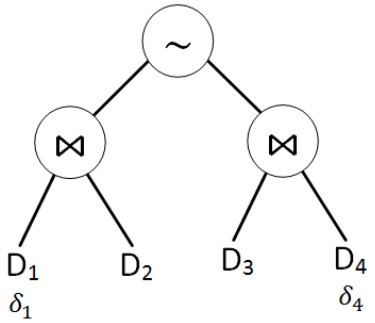

Figure 5.2: Concurrent data increments from two data containers (δ_1, δ_4)

According to the algorithms presented in Chapter 4, the increment expression to compute an increment data δ_i is formed in one of the following expressions:

$$f(D_1, \ldots, D_i, \ldots, D_k) \cup g_i(\delta_i, M_1, \ldots, M_j)$$
$$f(D_1, \ldots, D_i, \ldots, D_k) \sim g_i(\delta_i, M_1, \ldots, M_j)$$

Therefore, after a series of transformation processes we obtain increment expressions for individual data increments $\delta_1, \delta_2, \delta_3$ and δ_4 as follows:

$$
\begin{aligned}
\delta_1 :& f(D_1, D_2, D_3, D_4) \cup ((\delta_1 \bowtie D_2) \sim M_2) \\
=& f(D_1, D_2, D_3, D_4) \cup g_1 \\
\delta_2 :& f(D_1, D_2, D_3, D_4) \cup ((D_1 \bowtie \delta_2) \sim M_2) \\
=& f(D_1, D_2, D_3, D_4) \cup g_2 \\
\delta_3 :& f(D_1, D_2, D_3, D_4) \sim (\delta_3 \bowtie D_4) \\
=& f(D_1, D_2, D_3, D_4) \sim g_3 \\
\delta_4 :& f(D_1, D_2, D_3, D_4) \sim (D_3 \bowtie \delta_4) \\
=& f(D_1, D_2, D_3, D_4) \sim g_4
\end{aligned}
$$

where: $g_1 = ((\delta_1 \bowtie D_2) \sim M_2), g_2 = ((D_1 \bowtie \delta_2) \sim M_2), g_3 = (\delta_3 \bowtie D_4)$, $g_4 = (D_3 \bowtie \delta_4), M_2 = (D_3 \bowtie D_4)$.

To simplify notation, $g_i(\delta_i, M_1, \ldots, M_j)$ is written as g_i.

After all increment expressions for a single increment are obtained, we generate an online integration plan for every increment expression as follows:

$$
\begin{aligned}
d_1 :& \Delta_1 = (\delta_1 \bowtie D_2); \Delta_2 = (\Delta_1 \sim M_2); M_e = (M_e \cup \Delta_2); D_1 = (D_1 \cup \delta_1); \\
& M_1 = (M_1 \cup \Delta_1); \\
d_2 :& \Delta_3 = (D_2 \bowtie \delta_2); \Delta_4 = (\Delta_3 \sim M_2); M_e = (M_e \cup \Delta_4); D_2 = (D_2 \cup \delta_2); \\
& M_1 = (M_1 \cup \Delta_3); \\
d_3 :& \Delta_5 = (\delta_3 \bowtie D_4); M_e = (M_e \sim \Delta_5); D_3 = (D_3 \cup \delta_1); M_2 = (M_2 \cup \Delta_5); \\
d_4 :& \Delta_6 = (D_3 \bowtie \delta_4); M_e = (M_e \sim \Delta_6); D_4 = (D_4 \cup \delta_4); M_2 = (M_2 \cup \Delta_6);
\end{aligned}
$$

M_e is the final materialization, $M_1 = (D_1 \bowtie D_2)$ and $M_2 = (D_3 \bowtie D_4)$ are intermediate materializations. □

5.2 Processing of Concurrent Data Increments

Two data increments δ_i and δ_j in a data integration expression are called *concurrent data increments* if they arrive at the central site at about the same time. Processing of the concurrent data increments can be performed in two different ways:

1. **Serialization of increments**. In this approach, computation of concurrent

data increments is performed by processing one increment at a time. Serialization of concurrent increments δ_i and δ_j results in a sequence of processing data increments in the order of either δ_i followed by δ_j, or δ_j followed by δ_i. An algorithm to process each data increment is described in Chapter 4.

2. **A single increment expression for concurrent increments**. In this approach, all possible data containers which can be processed concurrently are identified. Then, the pre-processing stage generates all increment expressions including for concurrent data increments identified earlier. For concurrent data increments δ_i and δ_j, an increment expression is created using the following steps:

   (a) A data integration expression is transformed into increment expression of single increment for every data container δ_i and δ_j.

   (b) The final materialization in the increment expression for δ_j is replaced with an increment expression for δ_i. Then, data container D_i is replaced with $(D_i \cup \delta_i)$ and D_j with $(D_j \cup \delta_j)$ in the increment part.

   (c) Then, we transform the data integration expression in step (b) to obtain an increment expression which includes both increments δ_i and δ_j.

   At the end of the pre-processing stage, an online integration plan is obtained to process concurrent data increments.

5.3 Increment Expression for Concurrent Increments

In the **single increment expression for concurrent increments** approach, a sequence of transformations on a data integration expression is performed to produce an increment expression by considering all the increments at the participating data containers.

Example 5.2. *Increment expression for two concurrent increments*
Let δ_1 and δ_4 be concurrent data increments in a data integration expression $f(D_1, D_2, D_3, D_4) = (D_1 \bowtie D_2) \sim (D_3 \bowtie D_4)$ as shown in Example 5.1. Transformation of the data integration expression into an increment expression for concurrent data increments δ_1 and δ_4 can be performed in two ways. The first way is by transformation of the data integration expression by allowing two increments at once. The transformation is performed in the following steps:

1. D_1 and D_4 in the data integration expression are replaced with $(D_1 \cup \delta_1)$ and $(D_4 \cup \delta_4)$ respectively.

2. Then, XML algebra rules are applied to transform the data integration expression as follows:

$$
\begin{aligned}
\delta_{(1,4)} =& ((D_1 \cup \delta_1) \bowtie D_2) \sim (D_3 \bowtie (D_4 \cup \delta_4)) \\
=& ((D_1 \bowtie D_2) \cup (\delta_1 \bowtie D_2)) \sim ((D_3 \bowtie D_4) \cup (D_3 \bowtie \delta_4)) \\
=& ((D_1 \bowtie D_2) \sim ((D_3 \bowtie D_4) \cup (D_3 \bowtie \delta_4))) \cup ((\delta_1 \bowtie D_2)) \sim ((D_3 \bowtie D_4) \cup \\
& (D_3 \bowtie \delta_4)) \\
=& (((D_1 \bowtie D_2) \sim (D_3 \bowtie D_4)) \sim (D_3 \bowtie \delta_4)) \cup (((\delta_1 \bowtie D_2) \sim (D_3 \bowtie D_4)) \sim \\
& (D_3 \bowtie \delta_4)) \\
=& (\underbrace{((D_1 \bowtie D_2) \sim (D_3 \bowtie D_4))}_{f(D_1,D_2,D_3,D_4)} \cup \underbrace{((\delta_1 \bowtie D_2) \sim (D_3 \bowtie D_4)))}_{g_1} \sim \underbrace{(D_3 \bowtie \delta_4)}_{g_4}
\end{aligned}
\tag{5.2}
$$

From Equation 5.2, the increment expression for concurrent data increments δ_1 and δ_4 has two increment components; g_1 is an increment component for δ_1 and g_4 is for δ_4.

Another way is by using increment expressions for single increment obtained in Example 5.1 to generate an increment expression for concurrent increments δ_1 and δ_4. □

Let increment expressions for data increments δ_i and δ_j be as follows:

$$
\begin{aligned}
\delta_i &: \underbrace{f(D_1, \ldots, D_k)}_{materialization} \overset{\cup}{\sim} g_i(\delta_i, M_1, \ldots, M_j) \\
\delta_j &: \underbrace{f(D_1, \ldots, D_k)}_{materialization} \overset{\cup}{\sim} g_j(\delta_j, M_1, \ldots, M_j)
\end{aligned}
$$

We obtain an increment expression for concurrent increments δ_i and δ_j by replacing materialization part $(f(D_1, \ldots, D_k))$ with an increment expression of the other data increment such that it forms one of the following expressions:

$$
\begin{aligned}
&\delta_{(i,j)} : (f(D_1, \ldots, D_k) \overset{\cup}{\sim} g_j(\delta_j, M_1, \ldots, M_j)) \overset{\cup}{\sim} g_i(\delta_i, M_1, \ldots, M_j) \\
&\text{any data container } D_j \text{ in } g_i \text{ must be replaced with } (D_j \cup \delta_j) \\
&\delta_{(j,i)} : (f(D_1, \ldots, D_k) \overset{\cup}{\sim} g_i(\delta_i, M_1, \ldots, M_j)) \overset{\cup}{\sim} g_j(\delta_j, M_1, \ldots, M_j) \\
&\text{any data container } D_i \text{ in } g_j \text{ must be replaced with } (D_i \cup \delta_i)
\end{aligned}
$$

$\delta_{(i,j)}$ represents an increment expression for concurrent data increments δ_i and δ_j by replacing materialization in increment expression of δ_j with an increment expression δ_i.

Example 5.3. *Increment expression for concurrent increments by transformation of single increment expressions*
Let δ_1 and δ_4 be concurrent data increments in a data integration expression as shown in Example 5.1 ($f(D_1, D_2, D_3, D_4) = (D_1 \bowtie D_2) \sim (D_3 \bowtie D_4)$). Transformation of the data integration expression into an increment expression for concurrent data increments δ_1 and δ_4 is done step by step as follows:

1. The data integration expression is transformed to compute data increments δ_1 and δ_4, and we get an increment expression:

$$\delta_1 : f(D_1, D_2, D_3, D_4) \cup ((\delta_1 \bowtie D_2) \sim M_2) = f(D_1, D_2, D_3, D_4) \cup g_1 \quad (5.3)$$
$$\delta_4 : f(D_1, D_2, D_3, D_4) \sim (D_3 \bowtie \delta_4) = f(D_1, D_2, D_3, D_4) \sim g_4 \quad (5.4)$$

2. The transformation sequence is for an increment δ_1 followed by an increment δ_4. Therefore, we replace an expression $f(D_1, D_2, D_3, D_4)$ in Equation 5.4 with an expression in Equation 5.3 such that:

$$\begin{aligned}\delta_{(1,4)} : (f(D_1, D_2, D_3, D_4) \cup g_1) &\sim g_4 \\ &= (f(D_1, D_2, D_3, D_4) \cup ((\delta_1 \bowtie D_2) \sim M_2)) \sim (D_3 \bowtie \delta_4) \\ &= (f(D_1, D_2, D_3, D_4) \cup ((\delta_1 \bowtie D_2) \sim (D_3 \bowtie D_4))) \sim (D_3 \bowtie \delta_4)\end{aligned}$$

3. Then, D_4 is replaced with $(D_4 \cup \delta_4)$, and allow an application of XML algebra rules as shown in the following equations:

$$\begin{aligned}\delta_{(1,4)} : (f(D_1, D_2, D_3, D_4) \cup ((\delta_1 \bowtie D_2) \sim (D_3 \bowtie D_4))) &\sim (D_3 \bowtie \delta_4) \\ &= (f(D_1, D_2, D_3, D_4) \cup ((\delta_1 \bowtie D_2) \sim (D_3 \bowtie (D_4 \cup \delta_4)))) \sim (D_3 \bowtie \delta_4) \\ &= (f(D_1, D_2, D_3, D_4) \cup ((\delta_1 \bowtie D_2) \sim ((D_3 \bowtie D_4) \cup (D_3 \bowtie \delta_4)))) \sim (D_3 \bowtie \delta_4) \\ &= (f(D_1, D_2, D_3, D_4) \cup ((\delta_1 \bowtie D_2) \sim (D_3 \bowtie D_4) \sim (D_3 \bowtie \delta_4))) \sim (D_3 \bowtie \delta_4) \\ &= (f(D_1, D_2, D_3, D_4) \cup ((\delta_1 \bowtie D_2) \sim (D_3 \bowtie D_4)) \sim (D_3 \bowtie \delta_4)) \sim (D_3 \bowtie \delta_4) \\ &= (f(D_1, D_2, D_3, D_4) \cup ((\delta_1 \bowtie D_2) \sim (D_3 \bowtie D_4))) \sim (D_3 \bowtie \delta_4) \\ &= (f(D_1, D_2, D_3, D_4) \cup g_1) \sim g_4 \quad (5.5)\end{aligned}$$

To verify that the sequence of transformation does not affect the generated increment expression, we perform the same steps for transformation of the data

integration expression into an increment expression with transformation sequence δ_4 followed by δ_1 as follows:

1. The transformation sequence is for an increment δ_4 followed by δ_1. Therefore, the expression $f(D_1, D_2, D_3, D_4)$ in Equation 5.3 is replaced by an expression in Equation 5.4 such that:

$$\delta_{(4,1)} : (f(D_1, D_2, D_3, D_4) \sim g_4) \cup g_1 \tag{5.6}$$

2. To simplify the transformation, the increment expression is transformed one by one. An expression g_1 is transformed into g_1' by replacing all data containers D_4 in g_1 with $(D_4 \cup \delta_4)$. Then, g_4 is transformed into g_4' by replacing all data containers D_1 in g_4 with $(D_1 \cup \delta_1)$.

$$\begin{aligned} g_1' &= (\delta_1 \bowtie D_2) \sim (D_3 \bowtie (D_4 \cup \delta_4)) \\ &= (\delta_1 \bowtie D_2) \sim ((D_3 \bowtie D_4) \cup (D_3 \bowtie \delta_4)) \\ &= ((\delta_1 \bowtie D_2) \sim (D_3 \bowtie D_4)) \sim (D_3 \bowtie \delta_4) \\ &= (g_1 \sim g_4) \\ g_4' &= (D_3 \bowtie D4) = g_4 \end{aligned}$$

3. g_1 and g_4 in the Equation 5.6 are replaced by into g_1' and g_4' respectively. Then, the expression is transformed by application of XML algebra rules. The transformation is performed as follows:

$$\begin{aligned} \delta_{(4,1)} &: (f(D_1, D_2, D_3, D_4) \sim g_4') \cup g_1' \\ &= (f(D_1, D_2, D_3, D_4) \sim g_4) \cup (g_1 \sim g_4), \text{apply rule 5} \\ &= (f(D_1, D_2, D_3, D_4) \cup g_1) \sim g_4 \end{aligned} \tag{5.7}$$

From Equations 5.2, 5.5 and 5.7 it is evidence that transformation in any order of increment ends up with the same increment expressions. Then, it is concluded that the transformation of a data integration expression into an increment expression works for concurrent increments and can be performed by transformation of single increment expressions.

Using the same approach, increment expressions for other concurrent increments can be found as follows:

$$\delta_{(1,2)} : f(D_1, D_2, D_3, D_4) \cup ((\delta_1 \bowtie D_2) \sim M_2) \cup ((D_1 \bowtie \delta_2) \sim M_2) \cup ((\delta_1 \bowtie \delta_2) \sim M_2) \tag{5.8}$$

$$= f(D_1, D_2, D_3, D_4) \cup g_1 \cup g_2 \cup g_{(1,2)}$$

$$\delta_{(1,3)} : (f(D_1, D_2, D_3, D_4) \cup ((\delta_1 \bowtie D_2) \sim M_2)) \sim (\delta_3 \bowtie D_4) \tag{5.9}$$

$$= (f(D_1, D_2, D_3, D_4) \cup g_1) \sim g_3$$

$$\delta_{(2,3)} : (f(D_1, D_2, D_3, D_4) \cup ((D_1 \bowtie \delta_2) \sim M_2)) \sim (D_3 \bowtie \delta_4) \tag{5.10}$$

$$= (f(D_1, D_2, D_3, D_4) \cup g_2) \sim g_3$$

$$\delta_{(2,4)} : (f(D_1, D_2, D_3, D_4) \cup ((\delta_1 \bowtie D_2) \sim (D_3 \bowtie D_4))) \sim (D_3 \bowtie \delta_4) \tag{5.11}$$

$$: (f(D_1, D_2, D_3, D_4) \cup ((\delta_1 \bowtie D_2) \sim M_2)) \sim (D_3 \bowtie \delta_4) \tag{5.12}$$

$$= (f(D_1, D_2, D_3, D_4) \cup g_2) \sim g_4$$

$$\delta_{(3,4)} : f(D_1, D_2, D_3, D_4) \sim (\delta_3 \bowtie D_4) \sim (D_3 \bowtie \delta_4) \sim (\delta_3 \bowtie \delta_4) \tag{5.13}$$

$$= f(D_1, D_2, D_3, D_4) \sim g_3 \sim g_4 \sim g_{(3,4)}$$

where:

$g_1 = ((\delta_1 \bowtie D_2) \sim M_2); g_2 = ((D_1 \bowtie \delta_2) \sim M_2); g_3 = (\delta_3 \bowtie D_4); g_4 = (D_3 \bowtie \delta_4);$
$g_{(1,2)} = ((\delta_1 \bowtie \delta_2) \sim M_2); M_1 = (D_1 \bowtie D_2); M_2 = (D_3 \bowtie D_4);$

□

For concurrent data increments at two data containers, we generate a single increment expression which is in one of the expressions shown in Theorem 3.

Theorem 3. Let δ_h, δ_i be concurrent increments of data containers D_h, D_i. Any data integration expression $f(D_1, \ldots, D_h \cup \delta_h, D_i \cup \delta_i, \ldots, D_k)$ can be always transformed into one of the following equivalent expressions:

$$f(D_1, \ldots, D_h, D_i, \ldots, D_k) \cup g_h \cup g_i \cup g_{(h,i)} \tag{5.14}$$

$$(f(D_1, \ldots, D_h, D_i, \ldots, D_k) \cup g_h \cup g_i) \sim g_{(h,i)} \tag{5.15}$$

$$((f(D_1, \ldots, D_h, D_i, \ldots, D_k) \cup g_h) \sim g_i) \cup g_{(h,i)} \tag{5.16}$$

$$((f(D_1, \ldots, D_h, D_i, \ldots, D_k) \cup g_h) \sim g_i) \sim g_{(h,i)} \tag{5.17}$$

$$(f(D_1, \ldots, D_h, D_i, \ldots, D_k) \sim g_h) \cup g_i \cup g_{(h,i)} \tag{5.18}$$

$$((f(D_1, \ldots, D_h, D_i, \ldots, D_k) \sim g_h) \cup g_i) \sim g_{(h,i)} \tag{5.19}$$

$$((f(D_1, \ldots, D_h, D_i, \ldots, D_k) \sim g_h) \sim g_i) \cup g_{(h,i)} \tag{5.20}$$

$$((f(D_1, \ldots, D_h, D_i, \ldots, D_k) \sim g_h) \sim g_i) \sim g_{(h,i)} \tag{5.21}$$

where:
$g_h = g_h(\delta_h, M_1, \ldots, M_j)$, $g_i = g_i(\delta_i, M_1, \ldots, M_j)$ and $g_{hi} = g_{hi}(\delta_h, \delta_i, M'_1, \ldots, M'_j)$

Proof. Theorem 3:
To prove the theorem, we use a set of XML algebra rules which is shown in Chapter 3 to transform a data integration expression into an increment expression.

1. We extend an expression:
   $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) = f(D_1, \ldots, D_i, \ldots, D_k) \cup g_i(\delta_i, M_1, \ldots, M_j)$
   with a *union* ($\cup$) operator where $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ be the first argument and $(D_{k+1} \cup \delta_{k+1})$ be the second argument. The expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is transformed as follows:

   $=f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) \cup (D_{k+1} \cup \delta_{k+1})$
   $=(f(D_1, \ldots, D_i, \ldots, D_k) \cup g_i(\delta_i, M_1, \ldots, M_j)) \cup (D_{k+1} \cup \delta_{k+1})$
   we apply rule 3
   $=(f(D_1, \ldots, D_i, \ldots, D_k) \cup g_i(\delta_i, M_1, \ldots, M_j)) \cup D_{k+1} \cup \delta_{k+1}$
   we apply rule 3
   $=(f(D_1, \ldots, D_i, \ldots, D_k) \cup D_{k+1}) \cup g_i(\delta_i, M_1, \ldots, M_j) \cup \delta_{k+1}$
   $=f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \cup g_i(\delta_i, M_1, \ldots, M_j) \cup g_{k+1}(\delta_{k+1})$
   Since $M_a = h_a(D_1, \ldots, D_k) : a = 1, \ldots, j$, we are able to form:
   $M'_a = h'_a(D_1, \ldots, D_{k+1}) : a = 1, \ldots, k$. Then:
   $=f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \cup g_i(\delta_i, M'_1, \ldots, M'_k) \cup g_{k+1}(\delta_{k+1})$

2. We extend an expression:
   $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) = f(D_1, \ldots, D_i, \ldots, D_k) \sim g_i(\delta_i, M_1, \ldots, M_j)$
   with a *union* ($\cup$) operator where $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ be the first argument and $(D_{k+1} \cup \delta_{k+1})$ be the second argument. The expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is transformed as follows:

   $=f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) \cup (D_{k+1} \cup \delta_{k+1})$
   $=(f(D_1, \ldots, D_i, \ldots, D_k) \sim g_i(\delta_i, M_1, \ldots, M_j)) \cup (D_{k+1} \cup \delta_{k+1})$
   we apply rule 3
   $=(\underbrace{(f(D_1, \ldots, D_i, \ldots, D_k) \sim g_i(\delta_i, M_1, \ldots, M_j)) \cup D_{k+1}}_{\text{we apply rule 9}}) \cup \delta_{k+1}$
   $=((f(D_1, \ldots, D_i, \ldots, D_k) \cup D_{k+1}) \sim (g_i(\delta_i, M_1, \ldots, M_j) \sim D_{k+1})) \cup \delta_{k+1}$
   $=(f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \sim (g_i(\delta_i, M_1, \ldots, M_j) \sim D_{k+1})) \cup g_{k+1}(\delta_{k+1})$

Since $M_a = h_a(D_1, \ldots, D_k) : a = 1, \ldots, j$, we are able to form:
$M'_a = h'_a(D_1, \ldots, D_k, D_{k+1}) : a = 1, \ldots, k$. Then:
$=(f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \sim g_i(\delta_i, M'_1, \ldots, M'_j)) \cup g_{k+1}(\delta_{k+1})$

3. We extend an expression:
$f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) = f(D_1, \ldots, D_i, \ldots, D_k) \cup g_i(\delta_i, M_1, \ldots, M_j)$
with a *union* ($\cup$) operator where $(D_{k+1} \cup \delta_{k+1})$ be the first argument and $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ be the second argument. The expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is transformed as follows:

$=(D_{k+1} \cup \delta_{k+1}) \cup f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$
$=(D_{k+1} \cup \delta_{k+1}) \cup (f(D_1, \ldots, D_i, \ldots, D_k) \cup g_i(\delta_i, M_1, \ldots, M_j))$
we apply rule 3
$=(D_{k+1} \cup f(D_1, \ldots, D_i, \ldots, D_k)) \cup g_i(\delta_i, M_1, \ldots, M_j) \cup \delta_{k+1}$
$=f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \cup g_i(\delta_i, M_1, \ldots, M_j) \cup g_{k+1}(\delta_{k+1})$
$=f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \cup g_i(\delta_i, M_1, \ldots, M_j) \cup g_{k+1}(\delta_{k+1})$

4. We extend an expression:
$f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) = f(D_1, \ldots, D_i, \ldots, D_k) \sim g_i(\delta_i, M_1, \ldots, M_j)$
with a *union* ($\cup$) operator where $(D_{k+1} \cup \delta_{k+1})$ be the first argument and $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ be the second argument. The expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is transformed as follows:

$=(D_{k+1} \cup \delta_{k+1}) \cup f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$
$=(D_{k+1} \cup \delta_{k+1}) \cup (f(D_1, \ldots, D_i, \ldots, D_k) \sim g_i(\delta_i, M_1, \ldots, M_j))$
we apply rule 3
$=(\underbrace{D_{k+1} \cup (f(D_1, \ldots, D_i, \ldots, D_k) \sim g_i(\delta_i, M_1, \ldots, M_j))}_{\text{we apply rule 10}}) \cup \delta_{k+1}$
$=((D_{k+1} \cup f(D_1, \ldots, D_i, \ldots, D_k)) \sim (g_i(\delta_i, M_1, \ldots, M_j) \sim D_{k+1})) \cup$
$g_{k+1}(\delta_{k+1})$
$=(f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \sim (g_i(\delta_i, M_1, \ldots, M_j) \sim D_{k+1})) \cup g_{k+1}(\delta_{k+1})$
Since $M_a = h_a(D_1, \ldots, D_k) : a = 1, \ldots, j$, we are able to form:
$M'_a = h'_a(D_1, \ldots, D_{k+1}) : a = 1, \ldots, k$. Then:
$=(f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \sim g_i(\delta_i, M'_1, \ldots, M'_k)) \cup g_{k+1}(\delta_{k+1})$

5. We extend an expression:
   $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) = f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) \cup g_i(\delta_i, M_1, \ldots, M_j)$
   with a *join* ($\bowtie$) operator where $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ be the first argument and $(D_{k+1} \cup \delta_{k+1})$ be the second argument. The expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is transformed as follows:

   $=f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) \bowtie (D_{k+1} \cup \delta_{k+1})$
   $=(f(D_1, \ldots, D_i, \ldots, D_k) \cup g_i(\delta_i, M_1, \ldots, M_j)) \bowtie (D_{k+1} \cup \delta_{k+1})$,
   we apply rule 4
   $=((f(D_1, \ldots, D_i, \ldots, D_k) \cup g_i(\delta_i, M_1, \ldots, M_j)) \bowtie D_{k+1}) \cup$
   $((f(D_1, \ldots, D_i, \ldots, D_k) \cup g_i(\delta_i, M_1, \ldots, M_j)) \bowtie \delta_{k+1})$
   we apply rule 4
   $=((f(D_1, \ldots, D_i, \ldots, D_k) \bowtie D_{k+1}) \cup (g_i(\delta_i, M_1, \ldots, M_j) \bowtie D_{k+1})) \cup$
   $((f(D_1, \ldots, D_i, \ldots, D_k) \bowtie \delta_{k+1}) \cup (g_i(\delta_i, M_1, \ldots, M_j) \bowtie \delta_k))$
   we apply rule 4
   $=(f(D_1, \ldots, D_i, \ldots, D_k) \bowtie D_{k+1}) \cup (g_i(\delta_i, M_1, \ldots, M_j) \bowtie D_{k+1}) \cup$
   $(f(D_1, \ldots, D_i, \ldots, D_k) \bowtie \delta_{k+1}) \cup (g_i(\delta_i, M_1, \ldots, M_j) \bowtie \delta_{k+1})$
   we apply rule 3
   $=f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \cup g_i(\delta_i, M_1, \ldots, M_j, D_{k+1}) \cup$
   $g_{k+1}(\delta_{k+1}, M_1, \ldots, M_j) \cup g_{(i,k+1)}(\delta_{k+1}, \delta_i, M_1, \ldots, M_j)$
   Since $M_a = h_a(D_1, \ldots, D_k) : a = 1, \ldots, j$, we are able to form:
   $M'_a = h'_a(D_1, \ldots, D_k, D_{k+1}) : a = 1, \ldots, j$. Then:
   $=f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \cup g_i(\delta_i, M'_1, \ldots, M'_j) \cup g_{k+1}(\delta_{k+1}, M_1, \ldots, M_j) \cup$
   $g_{(i,k+1)}(\delta_{k+1}, \delta_i, M_1, \ldots, M_j)$

6. We extend an expression:
   $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) = f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) \sim g_i(\delta_i, M_1, \ldots, M_j)$
   with a *join* ($\bowtie$) operator where $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ be the first argument and $(D_{k+1} \cup \delta_{k+1})$ be the second argument. The expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is transformed as follows:

   $=f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) \bowtie (D_{k+1} \cup \delta_{k+1})$
   $=(f(D_1, \ldots, D_i, \ldots, D_k) \sim g_i(\delta_i, M_1, \ldots, M_j)) \bowtie (D_{k+1} \cup \delta_{k+1})$,
   we apply rule 7

$=((f(D_1,\ldots,D_i,\ldots,D_k)\bowtie(D_{k+1}\cup\delta_{k+1}))\sim$
$(g_i(\delta_i,M_1,\ldots,M_j))\bowtie(D_{k+1}\cup\delta_{k+1}))$
we apply rule 4
$=((f(D_1,\ldots,D_i,\ldots,D_k)\bowtie D_{k+1})\cup(f(D_1,\ldots,D_i,\ldots,D_k)\bowtie\delta_{k+1}))\sim$
$((g_i(\delta_i,M_1,\ldots,M_j)\bowtie D_{k+1})\cup(g_i(\delta_i,M_1,\ldots,M_j)\bowtie\delta_{k+1}))$
$=(f(D_1,\ldots,D_i,\ldots,D_k,D_{k+1})\cup f(D_1,\ldots,D_i,\ldots,D_k,delta_{k+1}))\sim$
$(g_i(\delta_i,M_1,\ldots,M_j,D_{k+1})\cup g_{(i,k+1)}(\delta_i,M_1,\ldots,M_j,\delta_{k+1}))$
we apply rule 6
$=((f(D_1,\ldots,D_i,\ldots,D_k,D_{k+1})\cup f(D_1,\ldots,D_i,\ldots,D_k,\delta_{k+1}))\sim$
$g_i(\delta_i,M_1,\ldots,M_j,D_{k+1}))\sim(g_i(\delta_i,M_1,\ldots,M_j,\delta_{k+1})$
$=((f(D_1,\ldots,D_i,\ldots,D_k,D_{k+1})\cup g_{k+1}(\delta_{k+1},D_1,\ldots,D_i,\ldots,D_k))\sim$
$g_i(\delta_i,M_1,\ldots,M_j,D_{k+1}))\sim g_{(i,k+1)}(\delta_i,M_1,\ldots,M_j,\delta_{k+1})$;
Since $M_a=h_a(D_1,\ldots,D_k):a=1,\ldots,j$, we are able to form:
$M'_a=h'_a(D_1,\ldots,D_k,D_{k+1}):a=1,\ldots,k$. Then:
$=((f(D_1,\ldots,D_i,\ldots,D_k,D_{k+1})\cup g_{k+1}(\delta_{k+1},M_1,\ldots,M_j))\sim$
$g_i(\delta_i,M'_1,\ldots,M'_k))\sim g_{(i,k+1)}(\delta_i,\delta_{k+1},M_1,\ldots,M_k)$

7. We extend an expression:
$f(D_1,\ldots,D_i\cup\delta_i,\ldots,D_k)=f(D_1,\ldots,D_i\cup\delta_i,\ldots,D_k)\cup g_i(\delta_i,M_1,\ldots,M_j)$
with a *join* ($\bowtie$) operator where $(D_{k+1}\cup\delta_{k+1})$ be the first argument and $f(D_1,\ldots,D_i\cup\delta_i,\ldots,D_k)$ be the second argument. The expression $f(D_1,\ldots,D_i\cup\delta_i,\ldots,D_k)$ is transformed as follows:

$=(D_{k+1}\cup\delta_{k+1})\bowtie f(D_1,\ldots,D_i\cup\delta_i,\ldots,D_k)$
$=(D_{k+1}\cup\delta_{k+1})\bowtie(f(D_1,\ldots,D_i,\ldots,D_k)\cup g_i(\delta_i,M_1,\ldots,M_j))$,
we apply rule 4
$=(D_{k+1}\bowtie(f(D_1,\ldots,D_i,\ldots,D_k)\cup g_i(\delta_i,M_1,\ldots,M_j)))\cup$
$(\delta_{k+1}\bowtie(f(D_1,\ldots,D_i,\ldots,D_k)\cup g_i(\delta_i,M_1,\ldots,M_j)))$,
we apply rule 4
$=((D_{k+1}\bowtie f(D_1,\ldots,D_i,\ldots,D_k))\cup(D_{k+1}\bowtie g_i(\delta_i,M_1,\ldots,M_j)))\cup$
$((\delta_{k+1}\bowtie f(D_1,\ldots,D_i,\ldots,D_k))\cup(\delta_{k+1}\bowtie g_i(\delta_i,M_1,\ldots,M_j)))$
we apply rule 4
$=(f(D_1,\ldots,D_i,\ldots,D_k,D_{k+1})\cup g_i(\delta_i,M_1,\ldots,M_j,D_{k+1}))\cup$

$(g_{k+1}(\delta_{k+1}, M_1, \ldots, M_j) \cup g_{(i,k+1)}(\delta_{k+1}, \delta_i, M_1, \ldots, M_j))$

$=f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \cup g_i(\delta_i, M_1, \ldots, M_j, D_{k+1}) \cup$

$g_{k+1}(\delta_{k+1}, M_1, \ldots, M_j) \cup g_{(i,k+1)}(\delta_{k+1}, \delta_i, M_1, \ldots, M_j)$

Since $M_a = h_a(D_1, \ldots, D_k) : a = 1, \ldots, j$, we are able to form:

$M'_a = h'_a(D_1, \ldots, D_k, D_{k+1}) : a = 1, \ldots, k$. Then:

$=f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \cup g_i(\delta_i, M'_1, \ldots, M'_j) \cup g_{k+1}(\delta_{k+1}, M_1, \ldots, M_j) \cup$

$g_{(i,k+1)}(\delta_i, \delta_{k+1}, M_1, \ldots, M_j)$

8. We extend an expression:
$f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) = f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) \sim g_i(\delta_i, M_1, \ldots, M_j)$
with a *join* ($\bowtie$) operator where $(D_{k+1} \cup \delta_{k+1})$ be the first argument and $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ be the second argument. The expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is transformed as follows:

$=(D_{k+1} \cup \delta_{k+1}) \bowtie f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$

$=(D_{k+1} \cup \delta_{k+1}) \bowtie (f(D_1, \ldots, D_i, \ldots, D_k) \sim g_i(\delta_i, M_1, \ldots, M_j))$,

we apply rule 4

$=(D_{k+1} \bowtie (f(D_1, \ldots, D_i, \ldots, D_k) \sim g_i(\delta_i, M_1, \ldots, M_j))) \cup$

$(\delta_{k+1} \bowtie (f(D_1, \ldots, D_i, \ldots, D_k) \sim g_i(\delta_i, M_1, \ldots, M_j)))$,

we apply rule 8):

$=((D_{k+1} \bowtie f(D_1, \ldots, D_i, \ldots, D_k)) \sim g_i(\delta_i, M_1, \ldots, M_j)) \cup$

$((\delta_{k+1} \bowtie f(D_1, \ldots, D_i, \ldots, D_k)) \sim g_i(\delta_i, M_1, \ldots, M_j)))$

$=(f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \sim g_i(\delta_i, M_1, \ldots, M_j)) \cup$

$(f(D_1, \ldots, D_i, \ldots, D_k, \delta_{k+1}) \sim g_i(\delta_i, M_1, \ldots, M_j))$

$=(f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \sim g_i(\delta_i, M_1, \ldots, M_j)) \cup$

$(g_{k+1}(D_1, \ldots, D_i, \ldots, D_k, \delta_{k+1}) \sim g_i(\delta_i, M_1, \ldots, M_j))$

we apply rule 5

$=(f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \cup g_{k+1}(D_1, \ldots, D_i, \ldots, D_k, \delta_{k+1}))$

$\sim g_i(\delta_i, M_1, \ldots, M_j)$

$=(f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \cup g_{k+1}(\delta_{k+1}, M_1, \ldots, M_j)) \sim$

$g_i(\delta_i, M_1, \ldots, M_j)$

9. We extend an expression:
$f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) = f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) \cup g_i(\delta_i, M_1, \ldots, M_j)$

with a *antijoin* ($\sim$) operator where $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ be the first argument and $(D_{k+1} \cup \delta_{k+1})$ be the second argument. The expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is transformed as follows:

$= f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) \sim (D_{k+1} \cup \delta_{k+1})$

we apply rule 6

$= ((f(D_1, \ldots, D_i, \ldots, D_k) \cup g_i(\delta_i, M_1, \ldots, M_j)) \sim D_{k+1}) \sim \delta_{k+1},$

we apply rule 5

$= ((f(D_1, \ldots, D_i, \ldots, D_k) \sim D_{k+1}) \cup (g_i(\delta_i, M_1, \ldots, M_j) \sim D_{k+1})) \sim \delta_{k+1},$

$= (f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \cup g_i(\delta_i, M_1, \ldots, M_j, D_{k+1})) \sim g_{k+1}(\delta_{k+1})$

Since $M_a = h_a(D_1, \ldots, D_k) : a = 1, \ldots, j$, we are able to form:

$M'_a = h'_a(D_1, \ldots, D_{k+1}) : a = 1, \ldots, k$. Then:

$= (f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \cup g_i(\delta_i, M'_1, \ldots, M'_j)) \sim g_{k+1}(\delta_{k+1}, M_1, \ldots, M_j)$

10. We extend an expression:
$f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) = f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) \cup g_i(\delta_i, M_1, \ldots, M_j)$
with a *antijoin* ($\sim$) operator where $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ be the first argument and $(D_{k+1} \cup \delta_{k+1})$ be the second argument. The expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is transformed as follows:

$= f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) \sim (D_{k+1} \cup \delta_{k+1})$

$= ((f(D_1, \ldots, D_i, \ldots, D_k) \sim g_i(\delta_i, M_1, \ldots, M_j)) \sim (D_{k+1} \cup \delta_{k+1})$

we apply rule 6

$= f(D_1, \ldots, D_i, \ldots, D_k) \sim (\underbrace{g_i(\delta_i, M_1, \ldots, M_j) \cup (D_{k+1} \cup \delta_{k+1})}_{\text{we apply rule 3}})$

$= f(D_1, \ldots, D_i, \ldots, D_k) \sim (D_{k+1} \cup (g_i(\delta_i, M_1, \ldots, M_j) \cup \delta_{k+1}))$

we apply rule 11

$= (f(D_1, \ldots, D_i, \ldots, D_k) \sim D_{k+1}) \sim (g_i(\delta_i, M_1, \ldots, M_j) \cup \delta_{k+1})$

we apply rule 6

$= ((f(D_1, \ldots, D_i, \ldots, D_k) \sim D_{k+1}) \sim g_i(\delta_i, M_1, \ldots, M_j)) \sim \delta_{k+1},$

$= (f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \sim g_i(\delta_i, M_1, \ldots, M_j)) \sim g_{k+1}(\delta_{k+1})$

$= (f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \sim g_i(\delta_i, M_1, \ldots, M_j)) \sim g_{k+1}(\delta_{k+1})$

11. We extend an expression:
$f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) = f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) \cup g_i(\delta_i, M_1, \ldots, M_j)$

with a *antijoin* ($\sim$) operator where $(D_{k+1} \cup \delta_{k+1})$ be the first argument and $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ be the second argument. The expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is transformed as follows:

$=(D_{k+1} \cup \delta_{k+1}) \sim f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$

$=(D_{k+1} \cup \delta_{k+1}) \sim (f(D_1, \ldots, D_i, \ldots, D_k) \cup g_i(\delta_i, M_1, \ldots, M_j))$,

we apply rule 5

$=(D_{k+1} \sim (f(D_1, \ldots, D_i, \ldots, D_k) \cup g_i(\delta_i, M_1, \ldots, M_j))) \cup$

$(\delta_{k+1} \sim (f(D_1, \ldots, D_i, \ldots, D_k) \cup g_i(\delta_i, M_1, \ldots, M_j)))$

we apply rule 6

$=((D_{k+1} \sim f(D_1, \ldots, D_i, \ldots, D_k)) \sim g_i(\delta_i, M_1, \ldots, M_j)) \cup$

$((\delta_{k+1} \sim f(D_1, \ldots, D_i, \ldots, D_k)) \sim g_i(\delta_i, M_1, \ldots, M_j)))$

$=(f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \sim g_i(\delta_i, M_1, \ldots, M_j)) \cup$

$(g_{k+1}(\delta_{k+1}, M_1, \ldots, M_j) \sim g_i(\delta_i, M_1, \ldots, M_j))$

we apply rule 5

$=(f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \cup g_{k+1}(\delta_{k+1}, M_1, \ldots, M_j)) \sim g_i(\delta_i, M_1, \ldots, M_j)$

12. We extend an expression:
$f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) = f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k) \sim g_i(\delta_i, M_1, \ldots, M_j)$
with an *antijoin* ($\sim$) operator where $(D_{k+1} \cup \delta_{k+1})$ be the first argument and $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ be the second argument. The expression $f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$ is transformed as follows:

$=(D_{k+1} \cup \delta_{k+1}) \sim f(D_1, \ldots, D_i \cup \delta_i, \ldots, D_k)$

$=(D_{k+1} \cup \delta_{k+1}) \sim (f(D_1, \ldots, D_i, \ldots, D_k) \sim g_i(\delta_i, M_1, \ldots, M_j))$

we apply rule 5

$=(D_{k+1} \sim (f(D_1, \ldots, D_i, \ldots, D_k) \sim g_i(\delta_i, M_1, \ldots, M_j))) \cup$

$(\delta_{k+1} \sim (f(D_1, \ldots, D_i, \ldots, D_k) \sim g_i(\delta_i, M_1, \ldots, M_j)))$

we apply rule 12

$=((D_{k+1} \sim f(D_1, \ldots, D_i, \ldots, D_k)) \cup (D_{k+1} \bowtie g_i(\delta_i, M_1, \ldots, M_j))) \cup$

$((\delta_{k+1} \sim f(D_1, \ldots, D_i, \ldots, D_k)) \cup (\delta_{k+1} \bowtie g_i(\delta_i, M_1, \ldots, M_j)))$

$=(f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \cup g_i(\delta_i, M_1, \ldots, M_j, D_{k+1})) \cup$

$(g_{k+1}(\delta_{k+1}, M_1, \ldots, M_j) \cup g_{(i,k+1)}(\delta_i, \delta_{k+1}, M_1, \ldots, M_j))$

Since $M_a = h_a(D_1, \ldots, D_k) : a = 1, \ldots, j$, we are able to form:
$M'_a = h'_a(D_1, \ldots, D_k, D_{k+1}) : a = 1, \ldots, k$. Then:
$= f(D_1, \ldots, D_i, \ldots, D_k, D_{k+1}) \cup g_i(\delta_i, M'_1, \ldots, M'_k) \cup$
$g_{k+1}(\delta_{k+1}, M_1, \ldots, M_j) \cup g_{(i,k+1)}(\delta_i, \delta_{k+1}, M_1, \ldots, M_j)$

□

For n concurrent data increments, a single increment expression is generated by the utilization of increment expressions for smaller concurrent data increments. Algorithm 5 shows an implementation to obtain an increment expression for concurrent data increments.

Algorithm 5 Transformation of a data integration expression into an increment expression for n number of concurrent increments

1: Generate increment expressions for every data container by considering a single increment. The transformation of the data integration expression is performed using the algorithms described earlier.
2: Get an increment expression for data container D_n. $\delta_n : f(D_1, \ldots, D_k) \overset{\cup}{\sim} g_n$
3: Replace materialization $(f(D_1, \ldots, D_k))$ in expression $\delta_n : f(D_1, \ldots, D_n) \overset{\cup}{\sim} g_n$ with an increment expression for concurrent increments $\delta_{(1,\ldots,(n-1))}$, such that the increment expression become $\delta_{(1,\ldots,n)} : (((f(D_1, \ldots, D_k) \overset{\cup}{\sim} g_1) \overset{\cup}{\sim} g_2) \overset{\cup}{\sim} \ldots \overset{\cup}{\sim} g_n)$.
4: **for each** increment expression g_x in an increment expression δ_n **do**
5: Replace D_i in g_x with $(D_i \cup \delta_i)$; $i = 1, \ldots, n$.
6: Transform g_x into g'_x such that it is in a form of Equation 4.6 or 4.7.
7: **end for**
8: Replace $g_x : i = 1, \ldots, (n-1)$ in the increment expression $\delta_{(1,\ldots,n)}$ with g'_x.

At the end of transformation process, an increment expression for an n concurrent data increments $\delta_1, \ldots, \delta_n$ is obtained in the form of:

$$\delta_{(1,\ldots,n)} : ((((f(D_1, \ldots, D_k) \overset{\cup}{\sim} g_1) \overset{\cup}{\sim} \ldots \overset{\cup}{\sim} g_n) \overset{\cup}{\sim} g_{(1,2)}) \overset{\cup}{\sim} \ldots \overset{\cup}{\sim} g_{(m,n)}) \overset{\cup}{\sim} g_{(1,\ldots,n)}$$

Example 5.4. *Transformation of a data integration expression into an increment expression for three concurrent increments*

Let δ_1, δ_3 and δ_4 be concurrent data increments in a data integration expression $f(D_1, D_2, D_3, D_4) = (D_1 \bowtie D_2) \sim (D_3 \bowtie D_4)$ as shown in Example 5.1. The increment expressions obtained for two concurrent increments in Example 5.3 is used. To show that the transformations can be performed in any order of increment processing, the data integration expression is transformed in all possible orders of the increment, and compare the transformation results in the following:

1. The first case is a transformation of a data integration expression for concurrent data increments in the order of: δ_1, δ_3, then δ_4.

(a) To get the increment expression, an increment expression for a single increment δ_4 is used, and $f(D_1, D_2, D_3, D_4)$ is replaced with an increment expression for concurrent increments δ_1 and δ_3. $\delta_{(1,3,4)}$ represents an expression for three concurrent increments from D_1, D_3, and D_4 respectively. Meanwhile, $\delta_{(1,3)}$ is an expression for two concurrent increments at D_1 and D_3.

$\delta_4 : f(D_1, D_2, D_3, D_4) \sim g_4$

$\delta_{(1,3,4)} : \delta_{(1,3)} \sim g_4$; The expression $\delta_{(1,3)}$ is obtained from Example 5.3

$= ((f(D_1, D_2, D_3, D_4) \cup g_1) \sim g_3) \sim g_4$

$= ((f(D_1, D_2, D_3, D_4) \cup ((\delta_1 \bowtie D_2) \sim M_2)) \sim (\delta_3 \bowtie D_4)) \sim (D_3 \bowtie \delta_4)$

$= ((f(D_1, D_2, D_3, D_4) \cup ((\delta_1 \bowtie D_2) \sim (D_3 \bowtie D_4))) \sim (\delta_3 \bowtie D_4)) \sim (D_3 \bowtie \delta_4)$

(b) To simplify the transformation, g_1 is transformed into g_1' and g_3 into g_3' one by one, by replacing D_4 with $(D_4 \cup \delta_4)$ in the data integration expression as follows:

$$
\begin{aligned}
g_1' &= (\delta_1 \bowtie D_2) \sim (D_3 \bowtie (D_4 \cup \delta_4)) \\
&= (\delta_1 \bowtie D_2) \sim ((D_3 \bowtie D_4) \cup (D_3 \bowtie g_4)) \\
&= (\delta_1 \bowtie D_2) \sim (D_3 \bowtie D_4)) \sim (D_3 \bowtie \delta_4) \\
&= (g_1 \sim g_4) \\
g_3' &= (\delta_3 \bowtie (D_4 \cup \delta_4)) \\
&= (\delta_3 \bowtie D_4) \cup (\delta_3 \bowtie \delta_4)) \\
&= (g_3 \cup g_{(3,4)})
\end{aligned}
$$

(c) Then, an increment expression of concurrent data increments δ_1, δ_3, and δ_4 is by replacement of g_1 and g_3 with g_1' and g_3' as follows:

$\delta_{(1,3,4)} : ((f(D_1, D_2, D_3, D_4) \cup g_1') \sim g_3') \sim g_4$

$= ((f(D_1, D_2, D_3, D_4) \cup (g_1 \sim g_4)) \sim (g_3 \cup g_{(3,4)})) \sim g_4$

$= ((((f(D_1, D_2, D_3, D_4) \cup g_1) \sim (g_4 \sim f(D_1, D_2, D_3, D_4))) \sim g_3) \sim g_{(3,4)}) \sim g_4$

Since $(A \sim B) \sim C = (A \sim C) \sim B$, then:

$= ((((f(D_1, D_2, D_3, D_4) \cup g_1) \sim (g_4 \sim f(D_1, D_2, D_3, D_4))) \sim g_4) \sim g_3) \sim g_{(3,4)})$

$= ((((f(D_1, D_2, D_3, D_4) \cup g_1) \sim ((g_4 \sim f(D_1, D_2, D_3, D_4)) \cup g_4)) \sim g_3) \sim g_{(3,4)})$

Since $((A \sim B) \cup A) = A$, then:

$$= ((((f(D_1, D_2, D_3, D_4) \cup g_1) \sim g_4) \sim g_3) \sim g_{(3,4)})$$
$$= ((((f(D_1, D_2, D_3, D_4) \cup g_1) \sim g_3) \sim g_4) \sim g_{(3,4)}) \quad (5.22)$$

Since the increment expression for concurrent increments δ_1, δ_3 is identical with the increment expression for concurrent increments δ_3, δ_1, then increment expression for $\delta_1, \delta_3, \delta_4$ is identical with the one for $\delta_3, \delta_1, \delta_4$.

2. Processing data increments in the order of: δ_1, δ_4, then δ_3.

   (a) To get the final result of an increment expression in this order, an increment expression for a single increment δ_4 is obtained and $f(D_1, D_2, D_3, D_4)$ is replaced with increment expression for concurrent increments δ_1 and δ_4.

   $\delta_3 : f(D_1, D_2, D_3, D_4) \sim g_3$

   $\delta_{(1,4,3)} : \delta_{(1,4)} \sim g_3$; The expression $\delta_{(1,4)}$ is obtained from Example 5.3

   $= ((f(D_1, D_2, D_3, D_4) \cup g_1) \sim g_4) \sim g_3$

   $= ((f(D_1, D_2, D_3, D_4) \cup ((\delta_1 \bowtie D_2) \sim M_2)) \sim (D_3 \bowtie \delta_4)) \sim (\delta_3 \bowtie D_4)$

   $= ((f(D_1, D_2, D_3, D_4) \cup ((\delta_1 \bowtie D_2) \sim (D_3 \bowtie D_4))) \sim (\delta_3 \bowtie D_4)) \sim$
   $(\delta_3 \bowtie D_4)$

   (b) To simplify the transformation, g_1 is transformed into g_1' and g_4 into g_4' one by one, by replacing D_3 with $(D_3 \cup \delta_3)$ in the data integration expression as follows:

$$\begin{aligned} g_1' &= (\delta_1 \bowtie D_2) \sim ((D_3 \cup \delta_3) \bowtie D_4) \\ &= (\delta_1 \bowtie D_2) \sim ((D_3 \bowtie D_4) \cup (\delta_3 \bowtie D_4)) \\ &= (\delta_1 \bowtie D_2) \sim (D_3 \bowtie D_4)) \sim (\delta_3 \bowtie D_4) \\ &= (g_1 \sim g_3) \\ g_4' &= ((D_3 \cup \delta_3) \bowtie \delta_4) \\ &= (D_3 \bowtie \delta_4) \cup (\delta_3 \bowtie \delta_4)) \\ &= (g_4 \cup g_{(3,4)}) \end{aligned}$$

   (c) Then, an increment expression of conccurent data increments δ_1, δ_3, and

δ_4 can be obtained by replacing g_1 and g_3 with g'_1 and g'_3 as follows:

$$\begin{aligned}
&\delta_{(1,3,4)} : ((f(D_1, D_2, D_3, D_4) \cup g'_1) \sim g'_4) \sim g_3 \\
&= ((f(D_1, D_2, D_3, D_4) \cup (g_1 \sim g_3)) \sim (g_4 \cup g_{(3,4)})) \sim g_3 \\
&= ((((f(D_1, D_2, D_3, D_4) \cup g_1) \sim (g_3 \sim f(D_1, D_2, D_3, D_4))) \sim g_4) \sim g_{(3,4)}) \sim g_3 \\
&\text{Since } (A \sim B) \sim C = (A \sim C) \sim B, \text{ then:} \\
&= ((((f(D_1, D_2, D_3, D_4) \cup g_1) \sim (g_3 \sim f(D_1, D_2, D_3, D_4))) \sim g_3) \sim g_4) \sim g_{(3,4)}) \\
&= ((((f(D_1, D_2, D_3, D_4) \cup g_1) \sim ((g_3 \sim f(D_1, D_2, D_3, D_4)) \cup g_3)) \sim g_4) \sim g_{(3,4)}) \\
&\text{Since } ((A \sim B) \cup A) = A, \text{ then:} \\
&= ((((f(D_1, D_2, D_3, D_4) \cup g_1) \sim g_3) \sim g_4) \sim g_{(3,4)}) \qquad (5.23)
\end{aligned}$$

Since the increment expression for concurrent increments δ_1, δ_3 is identical with the increment expression for concurrent increments δ_3, δ_1, increment expression for $\delta_1, \delta_3, \delta_4$ is identical with the one for $\delta_3, \delta_1, \delta_4$.

3. Processing data increments in the order of: δ_3, δ_4, then δ_1.

   (a) To produce the final result increment expression in this order, an increment expression for a single increment δ_1 is obtained and $f(D_1, D_2, D_3, D_4)$ is replaced with increment expression for concurrent increments δ_4 and δ_3.

$$\begin{aligned}
&\delta_1 : f(D_1, D_2, D_3, D_4) \cup ((\delta_1 \bowtie D_2) \sim M_2) \\
&\delta_{(3,4,1)} : \delta_{(3,4)} \cup g_1; \text{ The expression } \delta_{(3,4)} \text{ is obtained from Example 5.3} \\
&= (f(D_1, D_2, D_3, D_4) \sim g_3 \sim g_4 \sim g_{(3,4)}) \cup g_1
\end{aligned}$$

   (b) To simplify the transformation, g_3 is transformed into g'_3, g_4 into g'_4 and $g_{(3,4)}$ into $g'_{(3,4)}$ by replacing D_1 with $(D_1 \cup \delta_1)$. Then g_1 is transformed into g'_1 by replacing D_3 with $(D_3 \cup \delta_3)$ and D_4 with $(D_4 \cup \delta_4)$. The transformations are performed as follows:

$$\begin{aligned}
g'_3 &= (\delta_3 \bowtie D_4) = g_3 \\
g'_4 &= (D_3 \bowtie \delta_4) = g_4 \\
g'_1 &= ((\delta_1 \bowtie D_2) \sim ((D_3 \cup \delta_3) \bowtie (D_4 \cup \delta_4)) \\
&= ((\delta_1 \bowtie D_2) \sim ((D_3 \bowtie (D_4 \cup \delta_4)) \cup (\delta_3 \bowtie (D_4 \cup \delta_4)) \\
&= ((\delta_1 \bowtie D_2) \sim ((D_3 \bowtie D_4) \cup (D_3 \bowtie \delta_4)) \cup (\delta_3 \bowtie D_4) \cup (\delta_3 \bowtie \delta_4)) \\
&= ((\delta_1 \bowtie D_2) \sim (D_3 \bowtie D_4)) \sim (D_3 \bowtie \delta_4) \sim (\delta_3 \bowtie D_4) \sim (\delta_3 \bowtie \delta_4)
\end{aligned}$$

$$= (g_1 \sim g_4 \sim g_3 \sim g_{(3,4)})$$
$$g'_{(3,4)} = (\delta_3 \bowtie \delta_4) = g_{(3,4)}$$

(c) Then, an increment expression of concurrent data increments δ_3, δ_4, and δ_1 can be obtained by replacing g_1, g_3, g_4 and $g_{(3,4)}3$ with g'_1, g'_3, g'_4 and $g'_{(3,4)}$ as follows:

$$\delta_{(4,3,1)} : ((f(D_1, D_2, D_3, D_4) \sim g'_3 \sim g'_4 \sim g'_{(3,4)} \cup g'_1$$
$$\delta_{(4,3,1)} : ((f(D_1, D_2, D_3, D_4) \sim g_3 \sim g_4 \sim g_{(3,4)} \cup (g_1 \sim g_4 \sim g_3 \sim g_{(3,4)})$$
$$\delta_{(4,3,1)} : ((f(D_1, D_2, D_3, D_4) \sim (g_3 \cup g_4 \cup g_{(3,4)})) \cup (g_1 \sim (g_4 \cup g_3 \cup g_{(3,4)}))$$

apply rule 5

$$\delta_{(4,3,1)} : (f(D_1, D_2, D_3, D_4) \cup g_1) \sim (g_3 \cup g_4 \cup g_{(3,4)})$$
$$\delta_{(4,3,1)} : (f(D_1, D_2, D_3, D_4) \cup g_1) \sim g_3 \sim g_4 \sim g_{(3,4)} \quad (5.24)$$

Since the increment expression for concurrent increments δ_4, δ_3 is identical with the increment expression for concurrent increments δ_3, δ_4, then increment expression for $\delta_4, \delta_3, \delta_1$ and the one for $\delta_3, \delta_4, \delta_1$ are identical.

From Equations 5.22, 5.23 and 5.24 it can be concluded that the transformation to generate a single increment expression works for any number of concurrent increments. □

5.4 Integration Plan for Concurrent Increments

An online integration plan for concurrent increments is obtained in the same way as for a single data increment (see Section 4.4.2). Given an increment expression for concurrent data increments below:

$$\delta_{(1,\ldots,n)} : ((((f(D_1, \ldots, D_k) \overset{\cup}{\sim} g_1) \overset{\cup}{\sim} \ldots \overset{\cup}{\sim} g_n) \overset{\cup}{\sim} g_{(1,2)}) \overset{\cup}{\sim} \ldots \overset{\cup}{\sim} g_{(m,n)}) \overset{\cup}{\sim} g_{(1,\ldots,n)}$$

we transform this into an online integration plan $d_{(1,\ldots,n)}$ using the following steps:

1. First, all increment components $(g_i : i = 1, \ldots, n)$ of the increment expression are mapped, such that every simple expression $(\omega_i : i = 1, \ldots, j)$ in $g_i(\delta_i, M_1, \ldots, M_j) = \omega_j(\ldots(\omega_2(\omega_1(\delta_i, M_1), M_2)\ldots)M_j)$ is mapped into a corresponding step from the inner-most to the outer most XML algebraic operation.

2. Next, several steps to update a final materialization (M_e) are appended to the result of the increment computation ($M_e = M_e \cup g_i$ or $M_e = M_e \sim g_i$). Every increment part on an increment expression contributes to the final materialization update.

3. In the next step of processing, all data containers which contribute in concurrent data increments ($D_i : i = 1, \ldots, n$) are updated with ($D_i \cup \delta_i$).

4. Then, all intermediate materializations $M_a = h_a(D_1, \ldots, D_k)$ are identified which are affected by all concurrent increments. The identification process uses the materialization dependency table generated in the pre-processing stage.

5. These procedures are performed to compute every data integration expression for materialization $M_a = h_a(D_1, \ldots, D_k) : a = 1, \ldots, j$ identified, but without a step to update the data containers.

Example 5.5. *Online integration plan for concurrent data increments*
Let δ_1 and δ_4 be concurrent data increments in a data integration expression as shown in Example 5.1. In a **single increment expression for concurrent increments** approach we obtain the following increment expression:

$$\delta_{(1,4)} : (\underbrace{((D_1 \bowtie D_2) \sim (D_3 \bowtie D_4))}_{M_e} \cup \underbrace{((\delta_1 \bowtie D_2) \sim M_2)}_{g_1}) \sim \underbrace{(D_3 \bowtie \delta_4)}_{g_4}$$

where $M_2 = (D_3 \bowtie D_4)$

The above procedure is used to create an online integration plan $d_{(1,4)}$ step by step as follows:

1. First, all simple operations in g_1 are mapped:
   $p_1 : \Delta_1 = (\delta_1 \bowtie D_2)$
   $p_2 : \Delta_2 = (\Delta_1 \sim M_2)$

2. Then, all simple operations in g_2 all mapped:
   $p_3 : \Delta_3 = (D_3 \bowtie \delta_4)$

3. Next, two steps to update the final materialization M_e are added with resutls of g_1 and g_4:
   $p_4 : \Delta_4 = (M_e \cup \Delta_2)$
   $p_5 : M_e = (\Delta_4 \sim \Delta_3)$

4. Two steps to update data container D_1 and D_4 are added:
   $p_6 : D_1 = (D_1 \cup \delta_1)$
   $p_7 : D_4 = (D_4 \cup \delta_4)$

5. In the next step, intermediate materializations M_1 and M_2 are updated:
   $p_8 : M_1 = (M_1 \cup \Delta_2)$
   $p_9 : M_2 = (M_2 \cup \Delta_3)$

At the end of transformation, an online integration plan is obtained as follows:

$$\begin{aligned} d_{(1,4)} :& \Delta_1 = (\delta_1 \bowtie D_2); \Delta_2 = (\Delta_1 \sim M_2); \Delta_3 = (D_3 \bowtie \delta_4); \Delta_4 = (M_e \cup \Delta_2); \\ & M_e = (\Delta_4 \sim \Delta_3); D_1 = (D_1 \cup \delta_1); D_4 = (D_4 \cup \delta_4); M_1 = (M_1 \cup \Delta_2); \\ & M_2 = (M_2 \cup \Delta_3); \end{aligned} \tag{5.25}$$

□

Next we compare online integration plans between two approaches in processing concurrent increments.

Example 5.6. *Comparison of two approaches in processing concurrent increments* Let $d_{(1,4)}$ be an online integration plan for concurrent increment δ_1 and δ_4 as in Equation 5.25. Using **a single increment for concurrent data increments approach**, the online integration plan is shown in Figure 5.3:

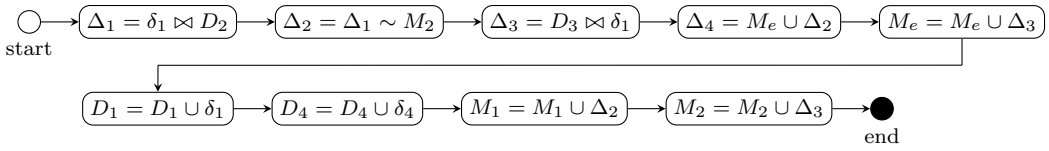

Figure 5.3: Execution of an online integration plan $d_{(1,4)}$

Meanwhile, in a **serialization approach** two online integration plans (d_1 and d_4) are executed. According to the Example 5.1, the execution plans is as follows:

$$\begin{aligned} d_1 :& \Delta_1 = (\delta_1 \bowtie D_2); \Delta_2 = (\Delta_1 \sim M_2); M_e = (M_e \cup \Delta_2); D_1 = (D_1 \cup \delta_1); \\ & M_1 = (M_1 \cup \Delta_1); \\ d_4 :& \Delta_6 = (D_3 \bowtie \delta_4); M_e = (M_e \sim \Delta_6); D_4 = (D_4 \cup \delta_4); M_2 = (M_2 \cup \Delta_6); \end{aligned}$$

The execution of online integration plans d_1 and d_4 is shown in Figure 5.4:

Figure 5.4 shows that in the **serialization aproach** the final materialization updates are processed at steps 4 and 8 each of which requires an operation to

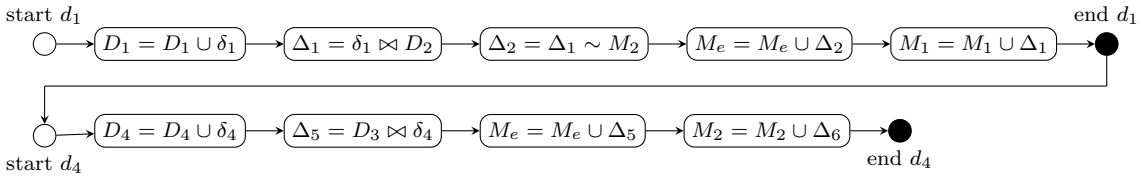

Figure 5.4: Execution of online integration plan d_1 and d_4 in the serialization approach

load the data from a secondary storage and then write back the results to that persistence storage.

Meanwhile, using **one increment expression for concurrent increments approach** as in Figure 5.3, we push operations to update the final materialization one after the other. This reduces the IO cost to retrieve materialization from a secondary storage as well as to store computation result back to the secondary storage. □

5.5 Parallel Processing of Online Data Integration Plans

Let δ_i and δ_j be concurrent data increments. The transformation of a data integration expression to compute δ_i and δ_j produces one increment expression in one of the expressions in Equation 5.14 - 5.21. The increment expression for two concurrent data increments has three increment components g_h, g_i, and $g_{(h,i)}$, where $g_{(h,i)}$ is optional.

In this thesis, an assumption is made that the central site is equipped with a parallel processing configuration which consists of several processors, and shares storage and memory among processors. Parallel processing of concurrent data increments is achieved by sending each increment component (g_i) to an independent processor. Then, the individual processor executes an online integration plan for the specific increment component g_i, for further computation. The computation results of g_i are used to update the final materialization based on the increment expression created earlier. The materialization update processes are performed serially. Meanwhile, intermediate materialization and data container updates can be performed again in parallel fashion.

Assuming that the central site has a sufficient number of processors, Figure 5.5 shows how an increment expression $\delta_{(1,\ldots,n)}$ is computed in parallel fashion.

Processing of each increment part of an increment expression (g_i) by a single

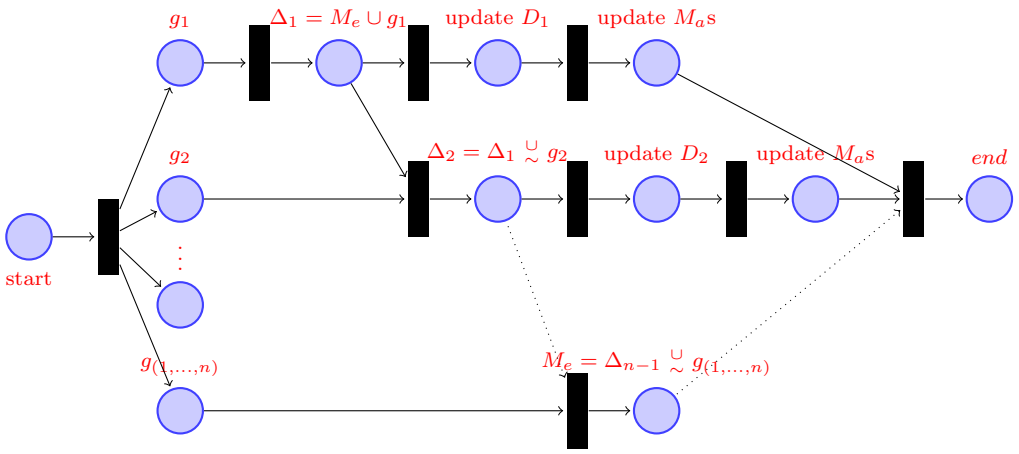

Figure 5.5: Parallel processing of increments by sending every independent increment component to a processor

independent processor is inefficient because an online integration plan for g_i may have a varying number of steps. Therefore, any processor which has completed its computation process must wait on other processing before the next step can be performed. Moreover, the number of concurrent data increments strictly depends on the number of processors at the central site.

To obtain better performance with a limited number of processors, we identify the process or task dependencies in the online integration plan and distribute the tasks among existing processors.

Example 5.7. *Parallel computation of increment expresions*
Parallel processing of online integration plans for concurrent increments δ_1 and δ_4 as in Example 5.5 is shown in Figure 5.7.

Let δ_1 and δ_4 be concurrent increments in the data integration expression in Example 5.1. In a **single increment expression for concurrent increments** approach the following increment expression is obtained:

$$\delta_{(1,4)} : (\underbrace{((D_1 \bowtie D_2) \sim (D_3 \bowtie D_4))}_{M_e} \cup \underbrace{((\delta_1 \bowtie D_2) \sim M_2)}_{g_1}) \sim \underbrace{(D_3 \bowtie \delta_4)}_{g_4}$$

where $M_2 = (D_3 \bowtie D_4)$

The online integration plan is shown step by step from p_1 to p_9 as follows:
$p_1 : \Delta_1 = (\delta_1 \bowtie D_2)$
$p_2 : \Delta_2 = (\Delta_1 \sim M_2)$
$p_3 : \Delta_3 = (D_3 \bowtie \delta_4)$
$p_4 : \Delta_4 = (M_e \cup \Delta_2)$
$p_5 : M_e = (\Delta_4 \sim \Delta_3)$

$p_6 : D_1 = (D_1 \cup \delta_1)$
$p_7 : D_4 = (D_4 \cup \delta_4)$
$p_8 : M_1 = (M_1 \cup \Delta_2)$
$p_9 : M_2 = (M_2 \cup \Delta_3)$
Then, processing can be performed as in the task dependency diagram shown in Figure 5.6.

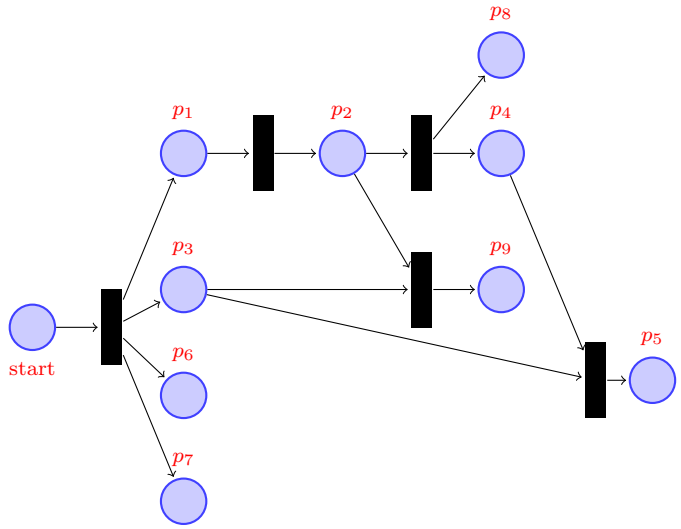

Figure 5.6: Task dependency diagram for processing of an increment expression $\delta_{(1,4)}$

If the central site has two processors, the execution of the online integration plan is modified based on their orders and utilization the processors' idle time. A process can be executed if it is independent or its dependent processes have completed. The execution plan is modified to that shown in Figure 5.7. □

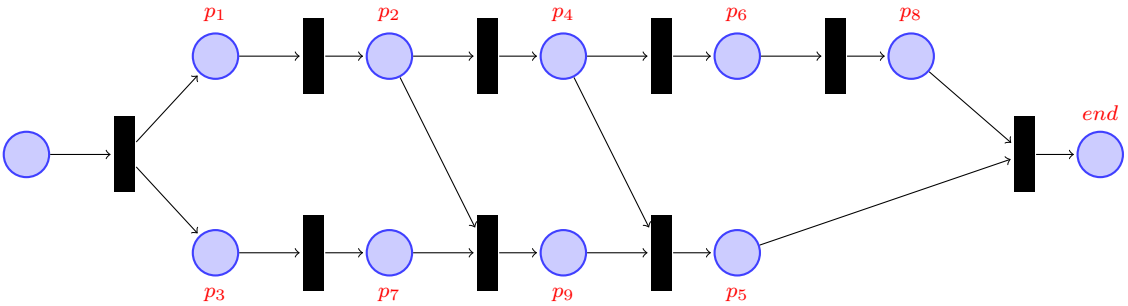

Figure 5.7: Modification of online integration plan to compute concurrent increments δ_1 and δ_4 in a parallel execution

Task-based parallel processing is not trivial and plays an important role in parallel processing. In this area, numerous optimization techniques have been

designed to optimize parallel processing. They can be directly applied to this problem, but are excluded from this thesis.

5.6 Scheduling of Online Integration Plans

We extend the scheduling system described in Chapter 4 to work on concurrent processing of increments. The scheduling system has two phases. The purpose of the **first phase** is to obtain a sequence of data increments in sorted priority labels. It is performed in the same way as described in Section 4.5. Meanwhile, the **second phase** intends to find data increments which can be computed in parallel mode; it is performed at the execution time.

Depending on the current data increment to process, concurrent processing is obtained by finding data increments in the following circumstances:

1. They share the same materialization to compute with. A materialized data usually has a larger size than a data container. Therefore, a process to load an intermediate materialization from a persistent storage into main memory requires an expensive IO cost.

2. They share the same materialization to update. The operation to update a materialized data requires a process to retrieve it from secondary storage and then write the update back to it. These operations are IO cost expensive, and the costs can be reduced if we perform operations to update a particular materialization one after the other. This strategy will minimize the IO costs to load intermediate materialization from and write it back to secondary storage.

Data increments which satisfy any conditions above are classified as type 2 (see Section 4.5.3).

The dynamic scheduling algorithm proposed in the previous chapters is modified to optimize processing of concurrent data increments. It modifies the order of processing data increments such that processing of increments which compute or update the same materializations are taken for processing in a parallel mode. Algorithm 6 shows an example dynamic scheduling algorithm for concurrent data increments.

Example 5.8. *Dynamic scheduling for concurrent data increments*
Let a sequence of increment data $\delta_{1a} \leftarrow \delta_{1b} \leftarrow \delta_2 \leftarrow \delta_{3a} \leftarrow \delta_4 \leftarrow \delta_{5a} \leftarrow \delta_{3b} \leftarrow \delta_{5b}$ arrives at the central site for an integration expression as in Figure 4.6. The scheduling system for concurrent increment data is performed as follows:

Algorithm 6 Dynamic scheduling for concurrent data increments

```
while (not empty queue) do
    Get increment data from a sliding window;
    Sort increment data based on their priority labels;
    for each increment in (sorted increment data) do
        Get all data increments in Type 2 of d_i; {d_x} : x = 1, ..., n
        Prepare online integration plan for d_(i,{x});
        for each (step p_i in integration plan) do
            if (p_i=materialization (M_a) update then
                if (M_a is not used by next plan) then
                    Store the increments to the designated increment lists;
                    Defer step p_i;
                else
                    if not empty increment lists then
                        Flush the increment lists to M_a
                    end if
                    Execute step p_i;
                end if
            else
                Execute step p_i;
                if (p_i has no result) then
                    Terminate rest steps of current plan d_(i,{x});
                end if
            end if
        end for
    end for
    Get to the next sliding window;
end while
```

1. At the end of the **first phase**, a sequence of increment data $\delta_{3a} \leftarrow \delta_{3b} \leftarrow \delta_2 \leftarrow \delta_{1a} \leftarrow \delta_{1b} \leftarrow \delta_{5a} \leftarrow \delta_{5b} \leftarrow \delta_4$ is obtained in the sliding window.

2. The **second phase** of dynamic scheduling is performed in the following manner:

   (a) We find concurrent increments that can be processed together with the increment data δ_{3a}. Processing of δ_{3a} requires an intermediate materialization M_4 for computation. In the example, there is no data increment which requires M_4 in their plans.

   (b) We find data increments which have an online integration plan to update materializations M_1 and/or M_2. Online integration plans d_1 and d_2 satisfy this condition. Then, δ_2, and δ_{1a} are processed together.

Modification of the plan is made as:

$$
\begin{aligned}
d_{321} : & D_3 = (D_3 \cup \delta_3); D_2 = (D_2 \cup \delta_2); \Delta_1 = (\delta_3 \sim M_4); \Delta_2 = (\delta_2 \sim D_3); \\
& \Delta_3 = (D_1 \bowtie \Delta_2); M_2 = (M_2 \sim (D_1 \sim \delta_3)); D_1 = (D_1 \cup \delta_1); \\
& M_1 = (M_1 \sim \delta_3); M_1 = (M_1 \cup (\Delta_2 \sim \delta_3)); \Delta_4 = (\delta_1 \bowtie M_1); \\
& M_2 = (M_2 \cup \Delta_4); M_e = (M_e \sim (\Delta_1 \cup \Delta_3)); \\
& M_e = (M_e \cup (\Delta_4 \sim (\Delta_1 \cup \Delta_3));
\end{aligned}
$$

Then, at the first row we compute increment data δ_{3a}, δ_2 and δ_{1a} concurrently.

(c) The next concurrent increment will be δ_{3b}. Following the same stategy above we compute δ_{3b} and δ_{1b} together.

(d) The next increment to compute is δ_{5a} where the same intermediate materialization M_2 is shared in its computation. Therefore, δ_{5a} and δ_4 can be computed together.

(e) Lastly, the increment data δ_{5b} is computed.

At the end of the second phase, a modified schedule is produced as: $(\delta_{3a}, \delta_2, \delta_{1a}) \leftarrow (\delta_{3b}, \delta_{1b}) \leftarrow (\delta_{5a}, \delta_4) \leftarrow \delta_{5b}$, where increments in parentheses are computed in parallel.

□

Further dynamic scheduling strategies (i.e *early termination of plan*, and *procrastination of plan*) from Section 4.5.4 are applicable for parallel processing of increments.

Extension of an online data integration system presented in this thesis allows for processing some data increments in parallel. Multiple transformations of a data integration expression are performed for some data increments such that one increment expression and one online integration plan are generated at the end of the transformation process. Data containers which can be processed in parallel must satisfy the conditions specified in Section 5.2.

Although it helps to improve the performance of an online data integration system, parallel processing of data increments will also decrease in performance when data increments consist of a large number of XML elements. So far, processing of a data increment can be started after it is available at the central site as a complete XML document. In fact, not all elements in an XML document are needed

to trigger the computation of a data increment. Therefore, an improved online integration system might be obtained if: (1) processing large size XML documents is done in fragments; and (2) the minimal requirements of an XML document to be processed can be identified. In the next chapter the processing of a fragmented XML document is discussed.

Chapter 6

Processing of XML Fragments

The online data integration system proposed in the previous chapters considers an incoming of a complete XML document before processing of the increment can be started [49]. The waiting time to start processing of a data increment depends on how long the transmission time from the first byte arriving at the central site until all bytes of the document become available at the central site. This means that we need a longer waiting time for a large size data increment before it can be computed.

In a distributed multi-database system, the large documents from the remote sites might be sent to a central site in the form of XML fragments. The remote sites have the responsibility of splitting the large XML documents into a number of XML fragments, then send them to the central site. At the central site, the incoming XML fragments are combined to form the original documents for further processing. Although the XML fragments are sent in the same order they are disassembled, they might arrive at the central site in a random order as the communication network may have unpredictable delays.

In fact, processing a large data increment can be started as soon as the existing XML fragments have enough properties for processing the increment. Hence, we identify the minimum requirements for processing of an increment such that we can process a smaller unit of increment rather than a complete XML document. To enable processing on XML fragments we modify the algorithms described in the previous chapters.

The principles and assumptions to process XML fragments are described in Section 6.1. Section 6.2 covers fragmentation of XML documents. Then, XML algebraic operations on fragmented XML documents is described in Section 6.4. In Section 6.5 an online integration algorithm for fragmented XML documents is described.

6.1 Principles and Assumptions

Figure 6.1 shows how XML fragments are transferred by the remote sites and combined at a central site. During the transmission process, it may happen that some packages have significant delays in arriving at the central site. Therefore, some fragments might be unavailable at the moment when the central site is ready to process the incoming documents.

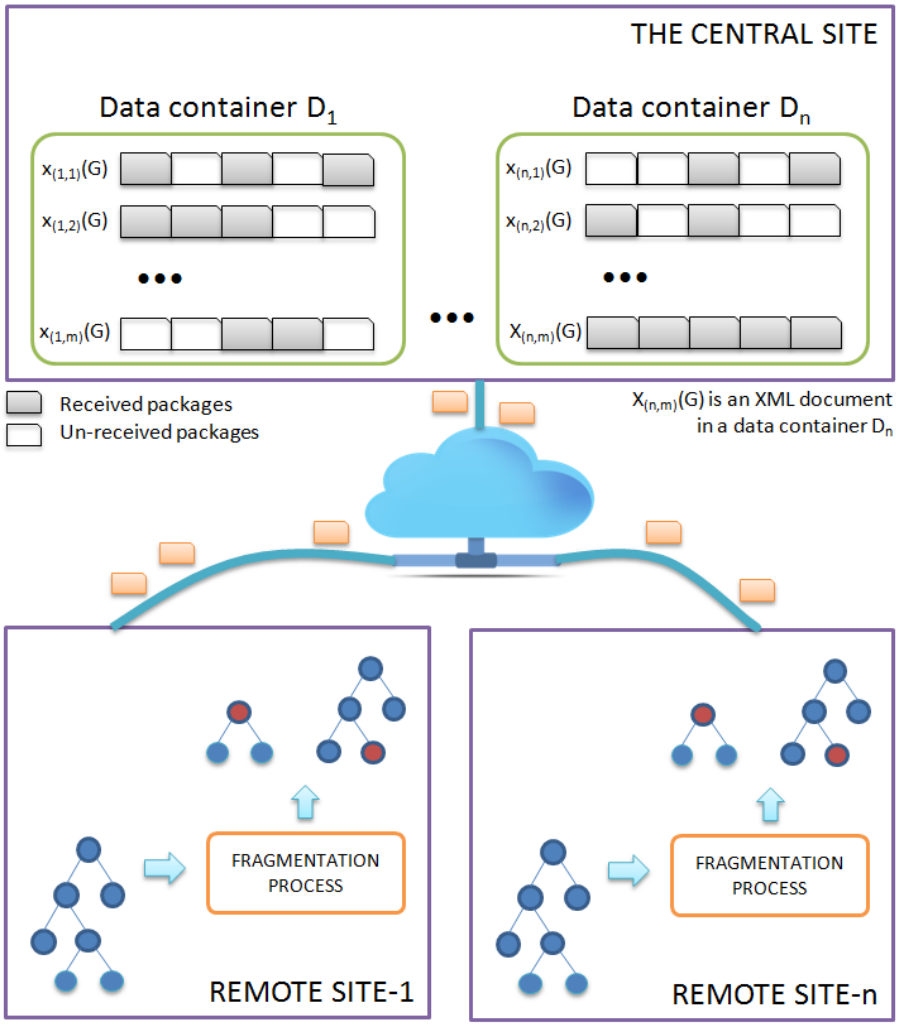

Figure 6.1: Transmission of XML fragments from the remote sites to a central site.

Figure 6.1 shows the data containers at the central site $(D_1, \ldots, D_n)$, where each of them contains a set of XML documents $x_{(i,j)} : i = 1, \ldots, n; j = 1, \ldots, m$. The complete XML documents have all their fragments available at the central site, otherwise they are classified as incomplete documents. $x_{(n,m)}$ in Figure 6.1 represents a complete XML document, while the other documents are classified as incomplete documents. In this thesis, both complete and incomplete documents are represented by fragmented XML documents.

We consider data increments which are transferred in the form of XML fragments. Then, the performance of an online integration system can be improved by processing a data increment in a smaller unit rather than a complete XML document. A data increment is processed as soon as all required XML fragments are available at the central site.

To enable processing of XML fragments in the online data integration system, the following assumptions are made:

1. The remote sites have the ability to disassemble XML documents into XML fragments with the characteristics described in the next section.

2. XML fragments from the remote sites are received by the central site in a random order.

3. The central site may reject XML fragments of a particular XML document which are no longer necessary for further computations.

4. The remote sites are highly autonomous.

Based on these assumptions, the following modifications to the online data integration system are made:

1. A fragmented XML document is a set of XML fragments.

2. The system has the same pre-processing as described in Chapter 4. The main difference is that transformation of a data integration is targeted for XML fragments instead of a complete XML document.

3. The remote sites are responsible for the fragmentation process, and the central site has the responsibility of combining the incoming XML fragments.

4. Every data container contains fragmented XML documents.

5. Attributes used in all the condition expressions are identified for every data container, and stored in an adequate list. The list is used to identify whether a set of XML fragments is ready for processing. Identification of attributes for condition expressions is discussed in Section 6.5.

Figure 6.1 shows that XML fragments are stored in a data container as a set of XML fragments.

6.2 Fragmented XML Documents

Operations on XML fragments allows us to compute incomplete XML documents, and to append their missing parts to the end of computations. At some point we may have results of XML algebraic operations as incomplete documents. With the incomplete documents scattered at some computation results, an algorithm is needed to locate the correct places to attach if the missing parts arrive at the central site.

In the fragmentation process, a unique identity of an XML document is copied and stamped along with its fragments to retain their origin document identity. Based on the assumption that every node in an XML document can be identified by a unique path (i.e path and index), an XML fragment is defined.

Definition 35. *An XML fragment is defined as* $\langle x_i(m_i), o, p, H\rangle$ *where* $x_i(m_i)$ *is an XML document as defined in Definition 11 and it represents the body of the XML fragment.* o *is an identity of the origin XML document where the fragment comes from, and* p *(hook) is a unique path where the XML fragment is located in the origin XML document.* H *is a set of paths to represent the missing the XML fragments (holes) in* $x_i(m_i)$*.*

Definition 36. *A fragmented XML document is a set of XML fragments* $\{\langle x_i(m_i), o_i, p_i, H_i\rangle : i = 1, \ldots, n\}$*.*

Definition 37. *Let* $x(m) = \{\langle x_i(m_i), o_i, p_i, H_i\rangle : i = 1, \ldots, n\}$ *be a fragmented XML document. A complete XML document is defined as a fragmented XML document where* $\underset{i}{\cup} p_i - \{$ `"xml"` $\} = \underset{i}{\cup} H_i$

Next, how the data containers are used to store the results of XML algebraic operations is considered. Figure 6.2 shows an illustration of how two *join* operations operate on fragmented XML documents.

A *join* operation on two XML documents described in Chapter 3 (see Definition 24) combines the XML documents by a *merge* operation on their ETGs. Let $r(l), s(m)$ and $t(n)$ be XML documents in Figure 6.2, where l=`xml[id](l')`, m=`xml[id](m')` and n=`xml[id](n')`. After processing of a data integration expression $f = (r(l) \bowtie (s(m) \bowtie t(n)))$, two XML documents $z(o)$ and $e(p)$ are obtained, where o =`xml[id](m'␣n')` and p =`xml[id](m'␣n'␣l')`. If $s(m)$ is a *fragmented XML document*, then $z(o)$ and $e(p)$ are *fragmented XML document*s.

Since *fragmented XML documents* are allowed to be processed in the online data integration system, the manner of storing the incoming XML fragments in a data container is covered in the next sections.

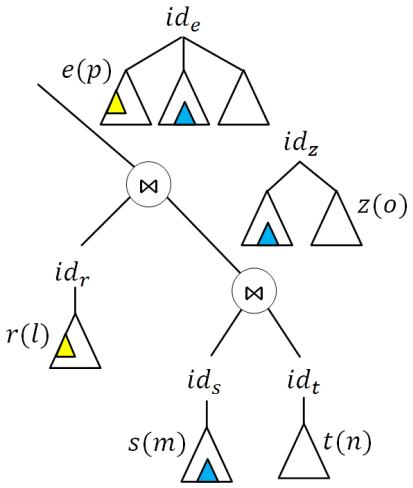

Figure 6.2: Join operation on XML fragments.

Example 6.1. *XML fragments in an XML document*
Let $x(m)$ be an XML document as shown in Figure 6.3 with an identifier `id="1"`. The XML document is disassembled into the six XML fragments which are shown in Figure 6.4, and their components shown in Table 6.1. Meanwhile, the body of XML fragments (a)-(f) in Figure 6.4 is shown as follows:

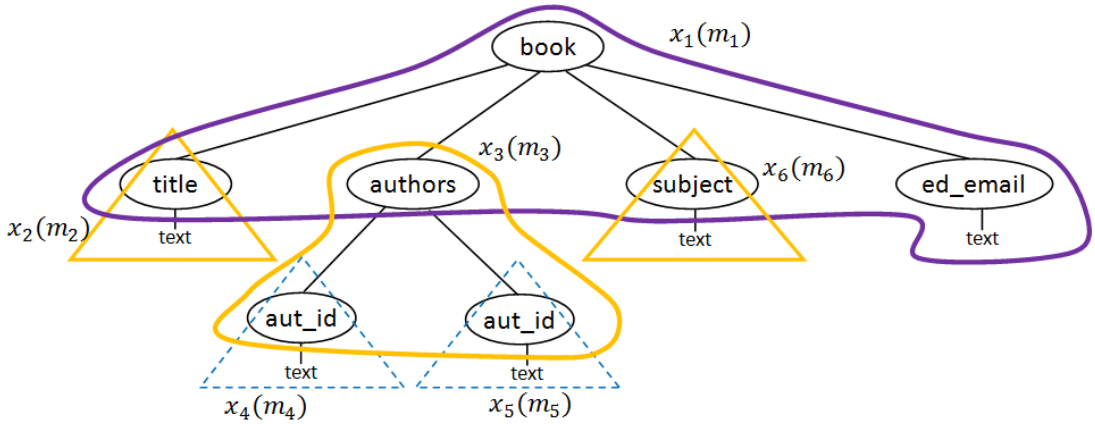

Figure 6.3: A complete XML document before fragmentation.

```
<xml id="1.1">
   <book isbn="9872347765" lang="EN">
      <title/>
      <authors/>
      <subject/>
      <ed_email>ross@gmail.com</ed_email>
   </book>
</xml>
```

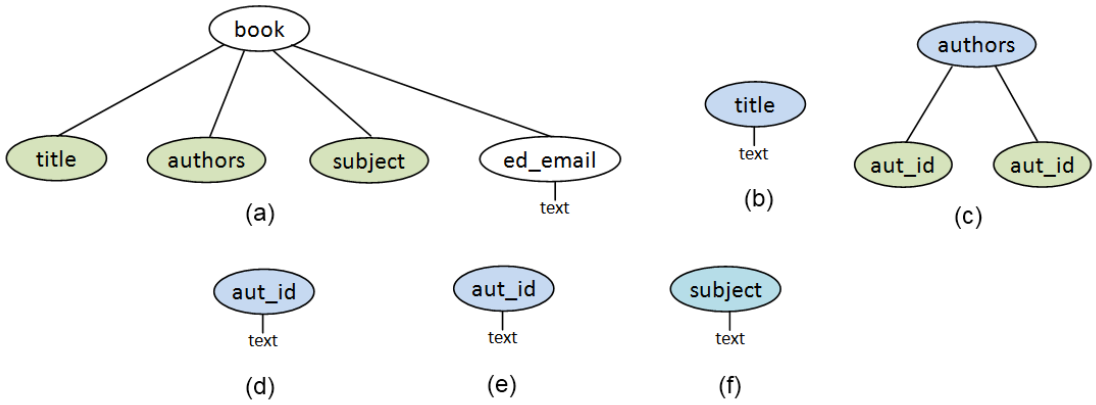

Figure 6.4: The body of XML fragments as result of fragmentation on XML document in Figure 6.3: **(a)** $\langle x_1(m_1), o_1, p_1, H_1\rangle$ **(b)** $\langle x_2(m_2), o_2, p_2, H_2\rangle$ **(c)** $\langle x_3(m_3), o_3, p_3, H_3\rangle$ **(d)** $\langle x_4(m_4), o_4, p_4, H_4\rangle$ **(e)** $\langle x_5(m_5), o_5, p_5, H_5\rangle$ **(f)** $\langle x_6(m_6), o_6, p_6, H_6\rangle$

Table 6.1: The components of XML fragments in Figure 6.4

XML Fragment	Origin id	Hook	Holes
(a)	o_1="1"	p_1="xml"	H_1={"xml/book/title", "xml/book/authors", "xml/book/subject"}
(b)	o_2="1"	p_2="xml/book/title"	H_2={}
(c)	o_3="1"	p_3="xml/book/authors"	H_3={"xml/book/authors/aut_id[1]", "xml/book/authors/aut_id[2]"}
(d)	o_4="1"	p_4="xml/book/authors/aut_id[1]"	H_4={}
(e)	o_5="1"	p_5="xml/book/authors/aut_id[2]"	H_5={}
(f)	o_6="1"	p_6="xml/book/subject"	H_6={}

```
<xml id="1.2">
   <title>XML</title>
</xml>

<xml id="1.3">
   <authors>
      <aut_id/>
      <aut_id/>
   </authors>
</xml>


<xml id="1.4">
   <aut_id>andy@yahoo.com</aut_id>
</xml>


<xml id="1.5">
   <aut_id>ben@yahoo.com</aut_id>
</xml>
```

```
<xml id="1.6">
   <subject>Data</subject>
</xml>
```

The ETGs of the XML fragments above are as the follows:

```
N={S,BOOK,TITLE,AUTHORS,AUT_EMAIL,SUBJECT,ED_EMAIL}
T={xml,book,title,authors,subject,ed_email}
A={id,isbn,lang}
P={S→xml[id](BOOK),BOOK→book[isbn lang](TITLE AUTHORS SUBJECT ED_EMAIL?),
   TITLE→title,AUTHORS→authors,SUBJECT→subject,ED_EMAIL→ed_email}

N={S,TITLE}
T={xml,title}
A={id}
P={S→xml[id](TITLE),TITLE→title}

N={S,AUTHORS,AUT_ID}
T={xml,authors,aut_id}
A={id}
P={S→xml[id](AUTHORS),AUTHORS→authors(AUT_ID*),AUT_ID→aut_id}

N={S,AUT_ID}
T={xml,aut_id}
A={id}
P={S→xml[id](AUT_ID),AUT_ID→aut_id}

N={S,AUT_ID}
T={xml,aut_id}
A={id}
P={S→xml[id](AUT_ID),AUT_ID→aut_id}

N={S,SUBJECT}
T={xml,subject}
A={id}
P={S→xml[id](SUBJECT),SUBJECT→subject}
```

□

Example 6.2. *A fragmented XML document*
Let $x(m)$ be an XML document where the original document and its XML fragments are as shown in Example 6.1. Let $\langle x_2(m_2), o_2, p_2, H_2\rangle$ and $\langle x_4(m_4), o_4, p_4, H_4\rangle$ be XML fragments which are available in a data container at the central site. Then, the XML fragments are stored in a data container in the form of a fragmented XML document $x(m)$ as follows:

$$x(m) = \{\langle x_2(m_2), o_2, p_2, H_2\rangle, \langle x_4(m_4), o_4, p_4, H_4\rangle\}$$

□

Example 6.3. *Join operations over fragmented XML documents*
Let $r(l) = \{\langle r_1(l_1), \texttt{"r"}, \texttt{"xml"}, H_1\rangle\}$, $s(m) = \{\langle s_2(m_2), \texttt{"s"}, p_2, H_2\rangle, \langle s_4(m_4), \texttt{"s"}, p_4, H_4\rangle\}$, and $t(n) = \{\langle t_1(n_1), \texttt{"t"}, \texttt{"xml"}, \{\}\rangle\}$ be fragmented XML documents in Figure 6.2. Let `"r"`, `"s"`, and `"t"` be the identities of the origin XML documents $r(l), s(m)$, and $t(n)$ respectively. After *join* operations, we obtain two fragmented XML document results $z(o)$ and $e(p)$ where:

$$z(o) = \{\langle s_2(m_2), \texttt{"s"}, p_2, H_2\rangle, \langle s_4(m_4), \texttt{"s"}, p_4, H_4\rangle, \langle t_1(n_1), \texttt{"t"}, \texttt{"xml"}, \{\}\rangle\}$$
$$e(p) = \{\langle r_1(l_1), \texttt{"r"}, \texttt{"xml"}, H_1\rangle, \langle s_2(m_2), \texttt{"s"}, p_2, H_2\rangle, \langle s_4(m_4), \texttt{"s"}, p_4, H_4\rangle, \langle t_1(n_1), \texttt{"t"}, \texttt{"xml"}, \{\}\rangle\}.$$

□

An XML fragment ($\langle x_i(m_i), o, p, H\rangle$) has the following characteristics:

1. An XML fragment body ($x_i(m_i)$) is a well-formed XML document.
2. An XML fragment has a component (o) to store the `id` attribute value of its original XML document where it comes from. Hence, all XML fragments from one XML document have the same o component values.
3. An XML fragment has a *hook* component (p) and a *hole* component (H) in order to enable the reconstruction of XML fragments into the origin XML document.
4. A *hook* component (p) is represented by a path, and determines where the XML fragment is located at the origin XML document. An XML fragment has exactly one *hook* component where *hook*=`"xml"` represents that the XML fragment has a root node of the origin XML document.
5. A *hole* (H) component is a set of paths where each path represents a missing part of the fragment. In the body of XML fragment, they are represented by empty elements.
6. An XML fragment may have zero or more missing fragments. An XML fragment which has an empty set of *hole* represents a complete sub-tree.

6.3 Fusion Operation on Extended Tree Grammars

The *merge* operation described in Chapter 3 is an operation to combine two ETGs into one ETG. The *merge* operation combines two XML document structures in a horizontal orientation, which means that two XML documents will be combined and placed at the same level in a document result. It is used to perform a *join* operation on two XML documents. In the processing of fragmented XML documents, the *join* and *merge* operations are redefined such that they are suitable for processing XML fragments.

Meanwhile, in order to assemble XML fragments we need another operation to combine two XML documents where one of them becomes a part of the other. A *fusion* operation on two ETGs is created as follows:

Definition 38. *Let* $G = (N_g, T_g, A_g, S_g, P_g)$ *and* $H = (N_h, T_h, A_h, S_h, P_h)$ *be ETGs,* $N_g \cap N_h \neq \emptyset$*,* Y *be a non terminal symbol, and* $y \in (N_g \cap N_h)$*. Let* `S→xml[id](Y)` *be a production rule for start symbol in* H*. Let* $p_g \in P_g$ *and* $p_h \in P_h$ *be production rules for a non terminal symbol* Y *in both ETGs. Fusion operation on two ETGs is denoted as* $F = G \oplus H$ *and is an operation that combines* G *and* H*, such that* $F = (N, T, A, S, P)$ *is an ETG where* $N = N_g \cup N_h$*,* $T = T_g \cup T_h$*,* $A = A_g \cup A_h$*, and* $P = P_g \cup P_h - \{p_g\}$*.*

Example 6.4. *An ETG result of fusion operation on two ETGs.*
Let G and H be ETG for XML fragments as in Figure 6.4 (a) and 6.4 (d). The *fusion* operation on G and H ($G \oplus H$) returns the ETG shown in Figure 6.7. The XML fragment result after a *fusion* operation is shown in Figure 6.8.

```
N={S,BOOK,TITLE,AUTHORS,AUT_EMAIL,SUBJECT,ED_EMAIL}
T={xml,book,title,authors,subject,ed_email}
A={id,isbn,lang}
P={S→xml[id](BOOK),BOOK→book[isbn lang](TITLE AUTHORS SUBJECT ED_EMAIL?),
   TITLE→title,AUTHORS→authors,SUBJECT→subject,ED_EMAIL→ed_email}
```

Figure 6.5: ETG of an XML fragment in Figure 6.4 (a)

□

6.4 XML Algebra Operations

XML algebraic operations for processing XML fragments are classified into three categories: (1) operations on XML fragments, (2) operations on fragmented XML

```
N={S,AUTHORS,AUT_ID}
T={xml,authors,aut_id}
A={id}
P={S→xml[id](AUTHORS),AUTHORS→authors(AUT_ID*),AUT_ID→aut_id}
```

Figure 6.6: ETG of an XML fragment in Figure 6.4 (b)

```
N={S,BOOK,TITLE,AUTHORS,SUBJECT,ED_EMAIL,AUT_ID}
T={xml,book,title,authors,subject,ed_email,aut_id}
A={id,isbn,lang}
P={S→xml[id](BOOK),BOOK→book[isbn lang](TITLE AUTHORS SUBJECT ED_EMAIL?),
   TITLE→title,AUTHORS→authors(AUT_ID*),SUBJECT→subject,AUT_ID→aut_id,
   ED_EMAIL→ed_email}
```

Figure 6.7: A result of *fusion* operation on two ETGs in Figure 6.5 and 6.6

documents, and (3) operations on data containers with fragmented XML document.

6.4.1 Hook Operation on XML Fragments

A *hook* operation combines two XML fragments which have matching *hook* and *hole* components to form a bigger XML fragment as result. After the result document is constructed, those two XML fragments are deleted. Definition 39 defines a *hook* operator to assemble two XML fragments.

Definition 39. *Let* $\langle x_i(m_i), o_i, p_i, H_i\rangle$, $\langle x_j(m_j), o_j, p_j, H_j\rangle$ *be XML fragments with ETG* G *and* H *respectively. Let* $o_i = o_j$ *and* $p_j \in H_i$. *A hook operation on two XML fragments is defined as* $\langle x_i(m_i), o_i, p_i, H_i\rangle \hookleftarrow \langle x_j(m_j), o_j, p_j, H_j\rangle = \langle x_r(m_r), o_r, p_r, H_r\rangle$. $x_r(m_r)$ *is an XML fragment result after a hook operation,* $o_r = o_i = o_j$, *and* $p_r = p_i$. $H_r = H_i \cup H_j - \{p_j\}$ *is a set of holes after a hook operation. The XML fragment result has an ETG* $F = G \oplus H$.

Example 6.5. *The result of hook operation on two XML fragments.*
Let x_{f1} and x_{f2} be XML fragments, where:

$$x_{f1} = \langle x_3(m_3), \texttt{"1"}, \texttt{"xml/book/authors"}, \{\texttt{"xml/book/authors/aut\_id[1]"}, \texttt{"xml/book/authors/aut\_id[2]"}\}\rangle$$

$$x_{f2} = \langle x_4(m_4), \texttt{"1"}, \texttt{"xml/book/authors/aut\_id[1]"}, \{\}\rangle$$

Hook operation $x_{f1} \hookleftarrow x_{f2}$ results to the following XML fragment:

$$\langle x_r(m_r), \texttt{"1"}, \texttt{"xml/book/authors"}, \{\texttt{"xml/book/authors/aut\_id[2]"}\}\rangle$$

```
<xml id="100">
   <book isbn="9872347765" lang="EN">
      <title/>
      <authors>
         <aut_id>andy@yahoo.com</aut_id>
         <aut_id/>
      </authors>
      <subject/>
      <ed_email/>
   </book>
</xml>
```

Figure 6.8: An XML document result of a fusion operation

□

Figure 6.9 shows an illustration of how an XML fragment is *hook*-ed to another XML fragment. In a tree structure, it is implemented by replacing a *hole* at an XML fragment (a) with a pointer of the other XML fragment (b).

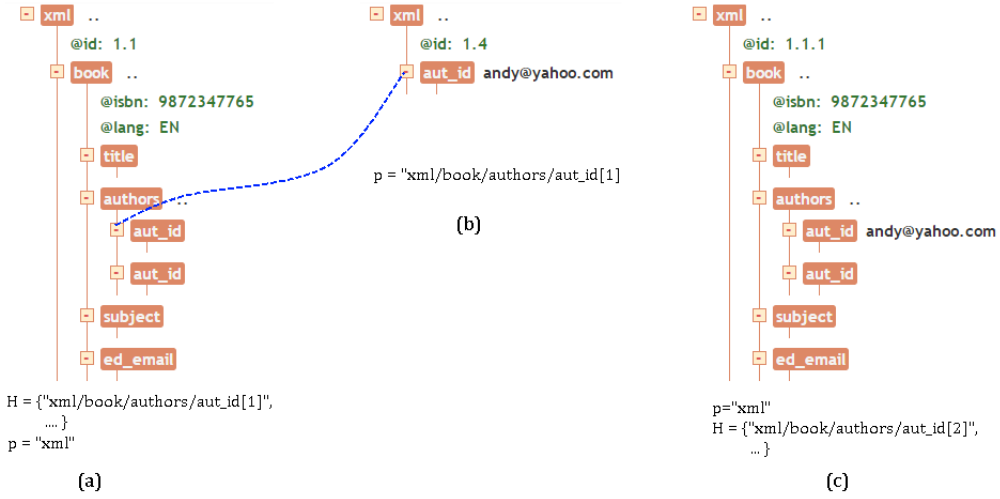

Figure 6.9: A hook operation in a tree structure.

6.4.2 Union Operation on Fragmented XML documents

At a certain stage of computation, we may want to combine some XML fragments into a fragmented XML document. This is achieved by a *union* operation on two sets of XML fragments. Having a fragmented XML document as a set of XML fragments allows us to perform a *union* operation which combines an incoming XML fragment with the existing ones.

Definition 40. *Let* $x(m) = \{\langle x_i(m_i), o_i, p_i, H_i\rangle : i = 1, \ldots, s\}$ *and* $y(n) = \{\langle y_j(n_j), o_j, p_j, H_j\rangle : j = 1, \ldots, t\}$ *be fragmented XML documents. Union operation on fragmented XML documents is defined as* $x(m) \cup y(n) = \{\langle z_k(m_k), o, p, H\rangle : \langle z_k(m_k), o, p, H\rangle \in x(m)$ *or* $\langle z_k(m_k), o, p, H\rangle \in y(n)\}$.

Example 6.6. *A union operation on two fragmented XML documents.*
Let $s(m) = \{\langle s_2(m_2), \texttt{"s"}, p_2, H_2\rangle, \langle s_4(m_4), \texttt{"s"}, p_4, H_4\rangle\}$, and $t(n) = \{\langle t_1(n_1), \texttt{"t"}, \texttt{"xml"}, \{\}\rangle\}$ be fragmented XML documents as in Figure 6.2. The *union* operation on fragmented XML documents $s(m)$ and $t(n)$, $(s(m) \cup t(n))$ produces a fragmented XML document as follows:

$$e(p) = \{\langle s_2(m_2), \texttt{"s"}, p_2, H_2\rangle, \langle s_4(m_4), \texttt{"s"}, p_4, H_4\rangle, \langle t_1(n_1), \texttt{"t"}, \texttt{"xml"}, \{\}\rangle\}$$

□

6.4.3 Defragmentation Procedure

At some point of processing, we might want to produce a bigger chunk of an XML fragment from smaller XML fragments in the fragmented XML documents. This is achieved by a *defragmentation* procedure, and utilizes *hook* and *hole* components of the XML fragments. If two XML fragments have a matching *hook* and *hole* values, they can be combined to form a bigger XML fragment. If all XML fragments are available in the fragmented XML document, then the *defragmentation* procedure produces a complete XML document in the form of a single set fragmented XML document.

The *defragmentation* procedure has two different operations:

1. **Multiple applications of the hook operation**. This is a defragmentation process on a fragmented XML document where at least one XML fragment has a *hole* component ($H \neq \emptyset$) and one XML fragment has a non root *hook* component ($p \neq$ `"xml"`). This defragmentation procedure applies multiple *hook* operations over all XML fragments in a fragmented XML document. When all XML fragments are available in the fragmented XML document, then it produces a complete XML document.

2. **A structural join operation**. This is a defragmentation process on a fragmented XML document where all XML fragments are complete ($p =$ `"xml"` and $H = \emptyset$). The defragmentation operation of this type combines the body of XML fragments by *merge* operations over their ETGs. Let $x(m) =$

$\{\langle x_i(m_i), o_x, \texttt{"xml"}, \{\}\rangle, \langle y_j(n_j), o_y, \texttt{"xml"}, \{\}\rangle\}$ be a fragmented XML document, and $m_i = \texttt{xml[id]}(m'_i)$, $n_j = \texttt{xml[id]}(n'_j)$. A defragmentation process on $x(m)$ results a fragmented XML document $z(p) = \{\langle z(p), \texttt{""}, \texttt{"xml"}, \{\}\rangle\}$ and $p = \texttt{xml[id]}(m'_i \lrcorner n'_j)$. o component for $z(p)$ is not needed since it is a complete XML document.

Algorithm 7 shows an implementation of the defragmentation operation.

Algorithm 7 Defragmentation of a fragmented XML document

Get a fragmented XML document $x(m) = \{\langle x_i(m_i), o_i, p_i, H_i\rangle : i = 1, \ldots, n\}$
i=1
if ($x(m)$ is a complete XML document) **then**
    j=i+1
    while (j ≤ n) **do**
        Perform $\langle x_r(m_r), o_r, p_r, H_r\rangle = \langle x_i(m_i \lrcorner m_j), \texttt{""}, \texttt{"xml"}, \{\}\rangle$
        Remove $\langle x_j(m_j), o_j, p_j, H_j\rangle$ from the fragmented XML document
        Copy $\langle x_r(m_r), o_r, p_r, H_r\rangle$ to $\langle x_i(m_i), o_i, p_i, H_i\rangle$
    end while
else
    while (i <n) **do**
        j=i+1
        while (j ≤ n) **do**
            if ($o_i = o_j$) **then**
                if ($p_j \in H_i$) **then**
                    Perform $\langle x_i(m_i), o_i, p_i, H_i\rangle \hookleftarrow \langle x_j(m_j), o_j, p_j, H_j\rangle$
                    Store the result in $\langle x_r(m_r), o_r, p_r, H_r\rangle$
                    Remove $\langle x_i(m_i), o_i, p_i, H_i\rangle$ from the fragmented XML document
                    Remove $\langle x_j(m_j), o_j, p_j, H_j\rangle$ from the fragmented XML document
                    Copy $\langle x_r(m_r), o_r, p_r, H_r\rangle$ to $\langle x_i(m_i), o_i, p_i, H_i\rangle$
                    n=n-1
                else
                    j=j+1
                end if
            else
                j=n
            end if
        end while
        i=i+1
    end while
end if

The defragmentation procedure in Algorithm 7 starts with identifying a pair of XML fragments which can be assembled. To simplify the process, we sort the XML fragments by their o and p components. The sorted XML fragments show that the position of XML fragments in a fragmented XML document represents their location at the origin XML document. If the XML fragments are sorted, then for two XML fragments $\langle x_i(m_i), o_i, p_i, H_i\rangle$ and $\langle x_j(m_j), o_j, p_j, H_j\rangle$ where i, j are the

element indexes and $i<j$, we can perform a *hook* operation $\langle x_i(m_i), o_i, p_i, H_i\rangle \hookleftarrow \langle x_j(m_j), o_j, p_j, H_j\rangle$ but not the opposite.

Example 6.7. *Defragmentation of a fragmented XML document*
Let a fragmented XML document have the following XML fragments as its elements:

$$
\begin{aligned}
s(m) = \{&\langle s_1(m_1), \texttt{"1"}, \texttt{"xml"}, \{\texttt{"xml/book/title","xml/book/authors","xml/book/subject"}\}\rangle,\\
&\langle s_2(m_2), \texttt{"1"}, \texttt{"xml/book/title"}, \{\}\rangle,\\
&\langle s_3(m_3), \texttt{"1"}, \texttt{"xml/book/authors"}, \{\texttt{"xml/book/authors/aut\_id[1]",}\\
&\texttt{"xml/book/authors/aut\_id[2]"}\}\rangle,\\
&\langle s_4(m_4), \texttt{"1"}, \texttt{"xml/book/authors/aut\_id[1]"}, \{\}\rangle,\\
&\langle s_5(m_5), \texttt{"1"}, \texttt{"xml/book/authors/aut\_id[2]"}, \{\}\rangle,\\
&\langle s_6(m_6), \texttt{"1"}, \texttt{"xml/book/subject"}, \{\}\rangle\}
\end{aligned}
$$

After the defragmentation process, the fragmented XML document becomes a single set of XML fragments $\{\langle s_r(m_r), \texttt{"1"}, \texttt{"xml"}, \{\}\rangle\}$ □

Online data integration of fragmented XML documents may produce some fragmented XML documents which have the same holes. They require the same XML fragments to fill the holes. When an incoming XML fragment matches the hole, it must be *hook*-ed to all of corresponding fragmented XML documents. There are three obstacles on processing incoming XML fragments into existing fragmented XML documents:

1. **Indicator of processing a fragmented XML document**.
   In a data container, there might be some fragmented XML documents that are ready for processing, but some are not. Therefore, we have to add a simple indicator to each of the fragmented XML document in order to identify the unprocessed ones. This has three indicator status: "not-ready" is for fragmented XML documents which are waiting to get minimal requirement to process; "ready" is when fragmented XML documents get enough properties for processing; "processed" is for fragmented XML documents in which parts of their fragments have been processed. We store the constraints in an adequate list and refer to the availability of certain elements which are used in the condition expression of the operations (φ).

2. **Algorithm to locate the fragmented XML documents in order to combine an incoming XML fragment**.
   Every XML fragment has a component to identify which XML document

they are fragmented from. This component is used to perform a defragmentation operation. However, when fragmented XML document results are scattered at some places, finding a particular fragmented XML documents in materializations is not trivial. In this thesis, a different approach is used to combine the missing fragments without searching to all the scattered fragmented XML documents.

3. **Cost of *hook* operations**.
   When an XML fragment arrives at a "processed" fragmented XML document, we have to assemble the particular fragment to all fragmented XML documents in the materializations which have the matching *hook-hole* components. This simply means that we require higher IO costs to update all materializations whenever one XML fragment arrives at the central site.

Among the three obstacles listed above, the biggest challenge is to reduce the costs to update all the fragmented XML documents. Therefore, we design an algorithm to process incoming XML fragments such that we can determine:

1. Where to place an incoming XML fragment.
2. When to combine an incoming XML fragment into any of existing fragmented documents in the data containers and materializations.
3. How to perform defragmentation process on the fragmented XML documents.

6.4.4 XML Algebra on Data Containers with Fragmented XML Documents

The missing XML fragments are placed in separate data containers and perform the operations to combine the XML fragments when necessary.

Definition 41. *Let $D(\mathcal{G}), D(\mathcal{H})$ be data containers of fragmented XML documents. Let $x(m) \in D(\mathcal{G}), x(m) = \{\langle x_i(m_i), o_i, p_i, H_i\rangle : i = 1, \ldots, s\}$ and $y(n) \in D(\mathcal{H})$, $y(n) = \{\langle y_j(n_j), o_j, p_j, H_j\rangle : j = 1, \ldots, t\}$. Let $O_i = \{o_i : \exists\langle x_i(m_i), o_i, p_i, H_i\rangle \in x(m) : i = 1, \ldots, s\}$, $O_j = \{o_j : j = 1, \ldots, t\}$ and $O_i \cap O_j \neq \emptyset$. A Minion (merge-union) operator is defined as $D(\mathcal{G}) \uplus D(\mathcal{H}) = \{z(l) : \exists x(m) \in D(\mathcal{G}), y(n) \in D(\mathcal{H}), z(l) = x(m) \cup y(n)\}$.*

In this approach, the following assumptions are made:

1. Every data container (D_i) argument of a data integration expression is divided into two data containers, a *bounded* data container (D_i^b) and a *rover* data container (D_i^r) such that $D = D^b \uplus D^r$. D^b (*bounded* data container) is used to store fragmented XML documents which are ready for processing. Meanwhile, XML fragments which have not been included in the computation are placed as elements of fragmented XML documents in a *rover* data container (D^r).

2. A new incoming XML fragment is placed as an element of a fragmented XML document in a data container D_i^r according to its origin XML document.

3. As soon as a fragmented XML document in D_i^r has minimum elements for computation, it is transferred to a *bounded* data container D_i^b, and processing of an increment is started. The sufficiency of a fragmented XML document is described in the next section.

4. The *minion* operation is performed on data containers of fragmented XML documents. For all materializations, the *minion* operation is performed at the end of computation, just before the final results are released to users.

Since we use a fragmented XML document to replace a complete XML document, most of the XML algebra operators described in Chapter 3 are applicable. Nevertheless, XML algebraic operations have to be re-defined because of two reasons:

1. The XML algebraic operations require a condition expression (φ) which no longer applies on a single XML document, but on a set of XML fragments.

2. For processing a fragmented XML document, a *join* operation combines two fragmented XML documents into one fragmented XML document using a *union* operation. Then, a *merge* operation is performed just before the results are sent to users.

Some XML algebraic operators require an examination of a condition expression (φ) on path expressions. For complete XML documents, path expressions refer to navigation paths from the root node of XML documents. Meanwhile, path evaluation on a fragmented XML document requires a process to discover a particular node in XML fragments which may not have its root element. Algorithm 8 finds a node specified by a path expression p in a fragmented XML document $x(m) = \{\langle x_i(m_i), o, p_i, H_i\rangle : i = 1, \ldots, n\}$.

Algorithm 8 Finding a path in a fragmented XML document

1: Sort XML fragments in $x(m)$ by length of p_i descending.
2: **for each** $\langle x_i(m_i), o_i, p_i, H_i\rangle$ in $x(m)$ **do**
3: Get *hook* component of $\langle x_i(m_i), o_i, p_i, H_i\rangle$
4: Remove index from p_i
5: **if** (Length(p_i) < Length(p)) **then**
6: Return **False**
7: **end if**
8: **if** ($p_i \subseteq p$) **then**
9: Get last part (node) of p and store as e
10: Set actual path p_a ="xml"+$(p - p_i)$+e
11: **if** (p_a exists in $x_i(m_i)$) **then**
12: Return **True**
13: **end if**
14: **end if**
15: **end for**

Example 6.8. *Finding a path in a fragmented XML document*
Let a fragmented XML document have the following XML fragments:

$s(m) = \{\langle s_1(m_1), \texttt{"1"}, \texttt{"xml"}, \{\texttt{"xml/book/title","xml/book/authors","xml/book/subject"}\}\rangle,$
$\langle s_2(m_2), \texttt{"1"}, \texttt{"xml/book/title"}, \{\}\rangle,$
$\langle s_3(m_3), \texttt{"1"}, \texttt{"xml/book/authors"}, \{\texttt{"xml/book/authors/aut\_id[1]",}$
$\texttt{"xml/book/authors/aut\_id[2]"}\}\rangle,$
$\langle s_4(m_4), \texttt{"1"}, \texttt{"xml/book/authors/aut\_id[1]"}, \{\}\rangle,$
$\langle s_5(m_5), \texttt{"1"}, \texttt{"xml/book/authors/aut\_id[2]"}, \{\}\rangle,$
$\langle s_6(m_6), \texttt{"1"}, \texttt{"xml/book/subject"}, \{\}\rangle\}$

Let φ : "xml/book/authors/aut_id='andy@yahoo.com'" be a condition expression, and p="xml/book/authors/aut_id" be a path expression. To evaluate a path expression p in $s(m)$, the following steps are performed:

1. The XML fragments are sorted on their *hook* components (p_i) in a descending order. We get the following order of XML fragments:
   $\langle s_5(m_5), \texttt{"1"}, \texttt{"xml/book/authors/aut\_id[2]"}, \{\}\rangle,$
   $\langle s_4(m_4), \texttt{"1"}, \texttt{"xml/book/authors/aut\_id[1]"}, \{\}\rangle,$
   $\langle s_6(m_6), \texttt{"1"}, \texttt{"xml/book/subject"}, \{\}\rangle,$
   $\langle s_2(m_2), \texttt{"1"}, \texttt{"xml/book/title"}, \{\}\rangle,$
   $\langle s_3(m_3), \texttt{"1"}, \texttt{"xml/book/authors"}, \{\texttt{"xml/book/authors/aut\_id[1]"},$
   $\texttt{"xml/book/authors/aut\_id[2]"}\}\rangle,$
   $\langle s_1(m_1), \texttt{"1"}, \texttt{"xml"}, \{\texttt{"xml/book/title","xml/book/authors",}$
   $\texttt{"xml/book/subject"}\}\rangle$

2. Take an XML fragment $\langle s_5(m_5), \texttt{"1"}, \texttt{"xml/book/authors/aut\_id[2]"}, \{\}\rangle$
3. Remove the index `"[2]"` from p_i
4. Check if the length of *hook* path `"xml/book/authors/aut_id"` is less than the length of `"xml/book/authors/aut_id"` path. It returns *true*.
5. Check if `"xml/book/authors/aut_id"` $\subseteq$ `"xml/book/authors/aut_id"`. It returns *true*.
6. Get $e =$ `"aut_id"`
7. Find a path `"xml/aut_id"` in an XML fragment: $\langle s_5(m_5), \texttt{"1"}, \texttt{"xml/book/authors/aut\_id[2]"}, \{\}\rangle$
8. The path is found but the value does not match the condition. Then the next XML fragment is processed.
9. Get the next XML fragment $\langle s_4(m_4), \texttt{"1"}, \texttt{"xml/book/authors/aut\_id[1]"}, \{\}\rangle$
10. Remove the index `"[1]"` from *hook* component (p_i) of the XML fragment.
11. Check if the length of `"xml/book/authors/aut_id"` is less than the length of `"xml/book/authors/aut_id"`. It returns *true*.
12. Check if (`"xml/book/authors/aut_id"` $\subseteq$ `"xml/book/authors/aut_id"`), It returns *true*.
13. Get $e =$ `"aut_id"`
14. Check path `"xml/aut_id"` in $\langle s_4(m_4), \texttt{"1"}, \texttt{"xml/book/authors/aut\_id[1]"}, \{\}\rangle$
15. Since the value matches the condition, then we return **true**, and process to find a path expression is ended

□

Definition 42. *Let $D(\mathcal{G})$ be a data container of fragmented XML documents, $x(m) \in D(\mathcal{G})$, $x(m) = \{\langle x_i(m_i), o, p_i, H_i\rangle : i = 1, \ldots, n\}$, and $x_f = \langle x_i(m_i), o_i, p_i, H_i\rangle$. Selection on $D(\mathcal{G})$ is a unary operator denoted by $\sigma_\varphi(D(\mathcal{G})) = \{x(m) : \exists x_f \in x(m)\ (f(x_f, \varphi) = true)\}$, where $f(x_f, \varphi)) \in \{true, false\}$, φ is a condition expression.*

Definition 43. *Let* $x(m) = \{\langle x_i(m_i), o_i, p_i, H_i\rangle : i = 1, \ldots, n\}$ *and* $y(n) = \{\langle y_j(n_j),\ o_j, p_j, H_j\rangle : j = 1, \ldots, m\}$ *be fragmented XML documents. Let* $x_f = \langle x_i(m_i), o_i, p_i,\ H_i\rangle$ *and* $y_f = \langle y_j(n_j), o_j,\ p_j, H_j\rangle$. *Join operation on fragmented XML documents is defined as* $x(m) \bullet_\varphi y(n) = x(m) \cup y(n) : \exists x_f \in x(m) \exists y_f \in y(n)$ *and* $f(x_f, y_f, \varphi) = true$. φ *is a condition expression and* f *is an* evaluation function *such that* $f(x_f, y_f, \varphi) \in \{true, false\}$.

Definition 44. *Let* $D(\mathcal{G}), D(\mathcal{H})$ *be data containers of fragmented XML documents. Join operation is defined as* $D(\mathcal{G}) \bowtie_\varphi D(\mathcal{H}) = \{z(o) : \exists x(m) \in D(\mathcal{G}), y(n) \in D(\mathcal{H}), z(o) = x(m) \bullet_\varphi y(n)\}$.

Definition 45. *Let* $D(\mathcal{G}), D(\mathcal{H})$ *be data containers of fragmented XML documents. Antijoin operator is defined as* $D(\mathcal{G}) \sim_\varphi D(\mathcal{H}) = \{x(m) : x(m) \in D(\mathcal{G})$ *and* $\forall y(n) \in D(\mathcal{H}) \neg\exists (x(m) \bullet_\varphi y(n))\}$, *where* φ *is a condition expression.*

6.4.5 Properties of the Minion Operation ($\uplus$)

Let D_i, D_j, D_k be data containers for fragmented XML documents. The *merge union* (*minion*) operation has the following properties:

1. The *minion* operation is commutative:
   $(D_i \uplus D_j) = (D_j \uplus D_i)$

2. The *minion* operation is associative:
   $D_i \uplus (D_j \uplus D_k) = (D_i \uplus D_j) \uplus D_k$

3. If the operation condition exists in D_i, then the *minion* operation is distributive over *selection* operation:
   $\sigma_\varphi(D_i \uplus D_j) = \sigma_\varphi(D_i) \uplus D_j$

4. If the operation condition exists in D_j, then the *minion* operation is distributive over *selection* operation:
   $\sigma_\varphi(D_i \uplus D_j) = D_i \uplus \sigma_\varphi(D_j)$

5. If properties for *join* operation condition exist in D_j, then the *minion* operation is distributive over *join* operation:
   $D_i \bowtie_\varphi (D_j \uplus D_k) = (D_i \bowtie_\varphi D_j) \uplus D_k$

6. If properties for *join* operation condition exist in D_k, then the *minion* operation is distributive over *join* operation:
   $D_i \bowtie_\varphi (D_j \uplus D_k) = (D_i \bowtie_\varphi D_k) \uplus D_j$.

7. If the operation condition exists in D_i, then the *minion* operation is distributive over *join* operation:
   $(D_i \uplus D_j) \bowtie_\varphi D_k = (D_i \bowtie_\varphi D_k) \uplus D_j$

8. If the operation condition exists in D_j, then the *minion* operation is distributive over *join* operation:
   $(D_i \uplus D_j) \bowtie_\varphi D_k = (D_j \bowtie_\varphi D_k) \uplus D_i$

9. The *minion* operation is distributive over the *union* operation:
   $D_i \cup (D_j \uplus D_k) = (D_i \cup D_j) \uplus D_k$
   $(D_i \uplus D_j) \cup D_k = (D_i \cup D_k) \uplus D_j$

10. If the operation condition exists in D_i, then the *minion* operation is distributive over *antijoin* operation:
    $(D_i \uplus D_j) \sim_\varphi D_k = (D_i \sim_\varphi D_k) \uplus D_j$

11. If the operation condition exists in D_j, then the *minion* operation is distributive over *antijoin* operation:
    $(D_i \uplus D_j) \sim_\varphi D_k = (D_j \sim_\varphi D_k) \uplus D_i$

12. The *minion* operation can reduce the *antijoin* operation:
    $D_i \sim_\varphi (D_j \uplus D_k) = (D_i \sim_\varphi D_j)$, if XML fragment D_j contains elements in operation condition (φ)
    $D_i \sim_\varphi (D_j \uplus D_k) = (D_i \sim_\varphi D_k)$, if XML fragment D_k contains elements in operation condition (φ)

6.5 Online Integration of XML Fragments

Online integration of XML fragments requires the management of the incoming XML fragments to allow the processing of fragmented XML documents as shown in Figure 6.10.

In order to start processing, an incomplete document must have enough properties required by query predicates of the XML operators. Online integration of XML fragments can be performed according to the following steps:

1. The pre-processing steps are similar to the procedures described in the previous chapters.

2. Data containers which are arguments of the data integration expression are

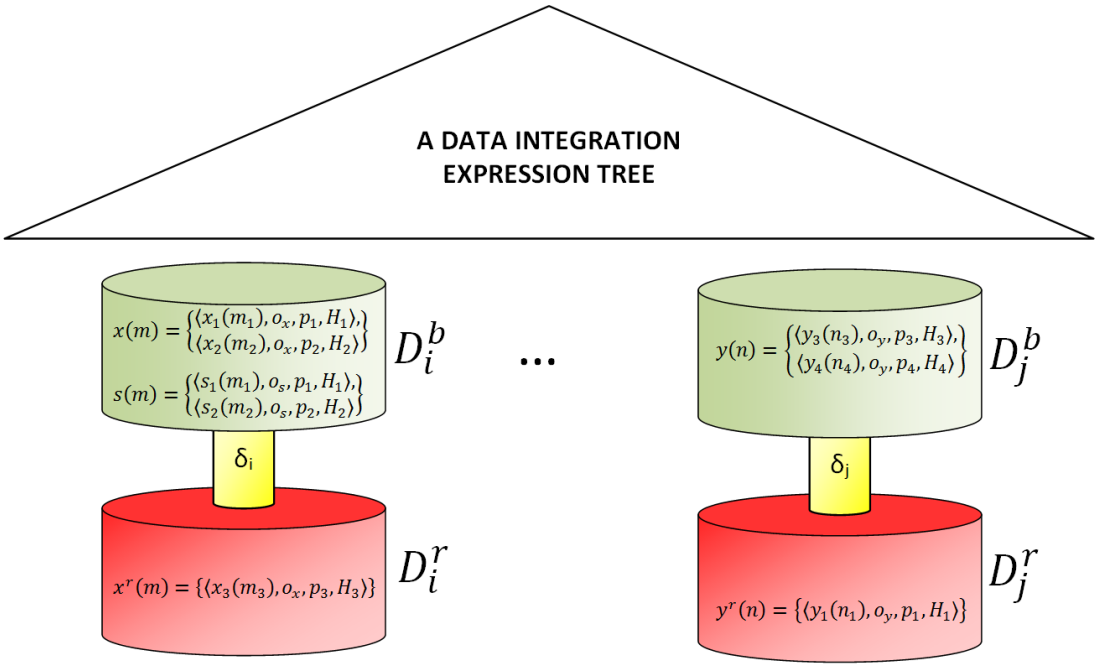

Figure 6.10: Fragmented XML documents in the data containers

split into two data containers D_i^b and D_i^r. A *bounded data container* D_i^b contains fragmented XML documents which are ready for computation. Meanwhile, a *rover data container* (D_i^r) contains fragmented XML documents which have not been computed.

3. An incoming XML fragment which arrives at the central site is placed in a *rover* data container (D_i^r) and is added to the corresponding fragmented XML document according to where it comes from.

4. If a fragmented XML document in the *rover* data container has enough properties for computation, we transfer it to a corresponding *bounded data container*. Incoming data at the *bounded* data containers triggers computation of a data integration expression.

5. Now a data container may be used in multiple operations and may use more than one XML elements in query predicates. Therefore, every data container has a list of necessary elements to identify which fragmented XML documents are ready for computation.

6. Before the results of a computation are sent to the end users, all the fragmented XML documents in the *rover* data containers are combined with their corresponding data containers and materializations by an application of the *minion* operations.

A fragmented XML document in the data container contains a set of XML fragments which have a uniform o component. The XML fragments are generated at the remote sites such that all XML fragments fragmented from an XML document retain the document's identity of the origin document. At the central site, an incoming XML fragment is placed in a fragmented XML document which has the same o component.

The process to place an incoming XML fragment into the existing fragmented XML document is performed in the following circumstances:

1. An incoming XML fragment does not match to any existing fragmented XML document. This means that the incoming XML fragment is the first fragment for the particular XML document. In this case, we create a new fragmented XML document and place the incoming XML fragment into it.

2. An incoming XML fragment matches an existing fragmented XML document. We append the incoming XML fragment to its corresponding fragmented XML document. Then we have an option to let the fragmented XML document be a set, or perform *defragmentation* procedure to build a bigger chunk.

3. An incoming XML fragment matches one of the existing fragmented XML documents but the properties are not enough to perform a *defragmentation* procedure on the fragmented XML document. In this case, we append the incoming XML fragment as an element of the fragmented XML document.

When dealing with XML fragments, a smaller unit of increment data can trigger the processing. Therefore, the online integration system needs to adjust the pre-processing phase to deal with XML fragments and requires the following information to be available:

1. All condition attributes (φ) are determined for all operations. For example, a data integration shown in Figure 6.11 has two operators which have conditions determined by φ_1 and φ_2.

2. All elements involved in the conditions are determined for every data container. For example, a data container D_1 involves in two operations where φ_1 requires an element in path p_1 and φ_2 requires an element in path p_2.

3. All available elements in the fragmented XML documents are determined. The existing nodes/elements are important to decide whether a fragmented XML document has enough properties for further processing.

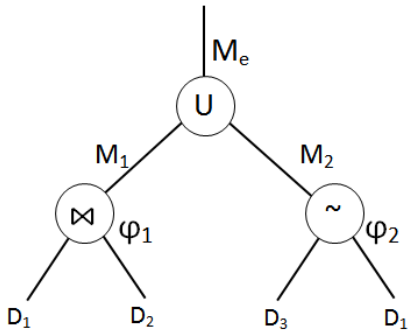

Figure 6.11: A syntax tree of a simple data integration expression

Now a data container might appear as several arguments in a data integration expression. Therefore, a data container must have a list of adequate properties. The structure of the adequate property list is shown in Figure 6.12:

Data Container :	Operation	Path	operator	Path or value

Figure 6.12: List of adequate properties

Example 6.9. *Adequate list for fragmented XML documents*
Let $f(D_1, D_2, D_3) = (D_1 \bowtie D_2) \cup (D_3 \sim D_1)$ be a data integration expression in Figure 6.11. A *join* operation $D_1 \bowtie_{(\varphi_1 \vee \varphi_2)} D_2$ has two condition expressions as follows:

$$\varphi_1 = \texttt{xml/book/authors/aut\_id[1]=//aut\_id}$$
$$\varphi_2 = \texttt{xml/book/authors/aut\_id[2]=//aut\_id}$$

XML paths `xml/book/authors/aut_id[1]` and `xml/book/authors/aut_id[2]` are unique locations of node elements of a fragmented XML document. Meanwhile, `//aut_id` is a path of a node element of a fragmented XML document in the data container D_2. Hence, an adequate list is generated as in Table 6.2. □

Table 6.2: An adequate lists of data containers for a data integration expression in Figure 6.11

Data container	**Operation**	**Path**	**Opr**	**Path or Value**
D_1	Operation 1	`xml/book/authors/aut_id[1]`	=	`//aut_id`
D_1	Operation 1	`xml/book/authors/aut_id[2]`	=	`//aut_id`
D_2	Operation 1	`//aut_id`	=	`xml/book/authors/aut_id`

Processing of fragmented XML documents may result in fragmented XML documents at the following parts:

1. **In the data containers**, where the particular incomplete documents have not been processed. Two data containers of fragmented XML documents are provided named *bounded* data containers (D^b) and *rover* data containers (D^r). All XML fragments arrive at the central site will be placed in a fragmented XML document in the corresponding *rover data container*. When the fragmented XML document has enough properties to compute, the *rover* data container transfers the fragmented XML document to the corresponding *bounded* data container. Thus, the *bounded* data containers contain all fragmented XML documents which are ready for computation. The remaining XML fragments of the processed fragmented XML document are stored in the *rover* data container.

2. **In a materialization (M_j)**, where the particular incomplete document has been computed.

3. **In a removed list (L_d)**, where the particular incomplete document does not meet criteria in the required condition (φ) and has no chance to meet the criteria.

6.5.1 Data Integration Expression

As described earlier, two data containers are used to replace a data container in Chapter 4, where $D_i = D_i^b \uplus D_i^r$. The *bounded* data containers get a new fragmented XML document transfered from *rover* data containers, which triggers the processing of a data increment.

To enable processing of an XML fragment, a data integration expression is transformed using the following steps:

1. All data containers D_i in a data integration expression are replaced such that $D_i = D_i^b \uplus D_i^r : i = 1 \ldots k$.

2. The data integration expression by multiple applications of *minion* properties is transformed such that all *rover* data containers are moved to the end of the computation process.

Example 6.10. *A data integration expression for XML fragments*
Let $D_1 = (D_1^b \uplus D_1^r), D_2 = (D_2^b \uplus D_2^r), D_3 = (D_3^b \uplus D_3^r), D_4 = (D_4^b \uplus D_4^r), D_5 = (D_5^b \uplus D_5^r), D_6 = (D_6^b \uplus D_6^r)$ be a data container for fragmented XML document. A data integration expression in Example 4.6 can be written as:

$$f(D_1, \ldots, D_6) = (D_1 \bowtie (D_2 \sim D_3)) \cup ((D_4 \bowtie D_5) \sim D_6)$$

$$
\begin{aligned}
&\text{we replace } D_1 \text{ with } D_1^b \uplus D_1^r \\
=&((D_1^b \uplus D_1^r) \bowtie (D_2 \sim D_3)) \cup ((D_4 \bowtie D_5) \sim D_6) \\
=&((D_1^b \bowtie (D_2 \sim D_3)) \uplus D_1^r) \cup ((D_4 \bowtie D_5) \sim D_6) \\
=&((D_1^b \bowtie (D_2 \sim D_3)) \cup ((D_4 \bowtie D_5) \sim D_6)) \uplus D_1^r \\
&\text{we replace } D_6 \text{ with } D_6^b \uplus D_6^r \\
=&((D_1^b \bowtie (D_2 \sim D_3)) \cup ((D_4 \bowtie D_5) \sim (D_6^b \uplus D_6^r))) \uplus D_1^r \\
=&((D_1^b \bowtie (D_2 \sim D_3)) \cup (((D_4 \bowtie D_5) \sim D_6^b) \uplus D_6^r)) \uplus D_1^r \\
=&((D_1^b \bowtie (D_2 \sim D_3)) \cup ((D_4 \bowtie D_5) \sim D_6^b)) \uplus D_1^r \\
&\text{If we transform all data containers to include data containers} \\
&\text{for XML fragment, we obtain:} \\
=&(((((D_1^b \bowtie (D_2^b \sim D_3^b)) \cup (D_4^b \bowtie D_5^b) \sim D_6^b) \uplus D_1^r) \uplus D_2^r) \uplus D_4^r) \uplus D_5^r
\end{aligned}
$$

To simplify notation, we use $D_1, D_2, D_3, D_4, D_5, D_6$ to replace $D_1^b, D_2^b, D_3^b, D_4^b$, D_5^b, D_6^b such that the data integration expression becomes:

$$(((((D_1 \bowtie (D_2 \sim D_3)) \cup ((D_4 \bowtie D_5) \sim D_6)) \uplus D_1^r) \uplus D_2^r) \uplus D_4^r) \uplus D_5^r.$$

□

Data increments which arrive at the central site will be handled according to the following steps:

1. Every XML fragment that arrives at the central site will be placed in a *rover* data container. It is combined as an element of a fragmented XML document according to its origin XML document.

2. An incoming XML fragment at a *rover* data container triggers an operation to examine sufficiency of the particular fragmented XML document for further processing. The sufficiency examination employs an adequate list which is prepared at the pre-processing stage.

3. When a fragmented XML document in a *rover* data container has enough properties, the fragmented XML document will be transferred into a *bounded* data container.

4. An incoming fragmented XML document in a *bounded* data container is used to trigger processing of a data increment. An incoming fragmented XML document at the *bounded* data container is treated in the same procedure as a data increment described in Chapter 4.

6.5.2 Increment Expression

In the next step, a data integration expression is transformed into an increment expression for every data container by application of XML algebra rules as described in Chapter 3.

As can be seen from the previous section, the *rover* data containers can be moved to the end of data integration expression, which means that the *defragmentation* process can be performed at the end of the computation.

Transformation of a data integration expression into an increment expression is performed using the same procedures as in the Chapter 4.

Example 6.11. *An increment expression for a data integration expression in Example 6.10.*
Let δ_1 be an increment data at a *bounded* data container D_1. The data integration expression $(((((D_1 \bowtie (D_2 \sim D_3)) \cup ((D_4 \bowtie D_5) \sim D_6)) \uplus D_1^r) \uplus D_2^r) \uplus D_4^r) \uplus D_5^r$ can be transformed as follows:

$$\begin{aligned}
f(D_1 \cup \delta_1, \ldots, D_6) =& ((((((D_1 \cup \delta_1) \bowtie (D_2 \sim D_3)) \cup ((D_4 \bowtie D_5) \sim D_6)) \uplus D_1^r) \uplus \\
& D_2^r) \uplus D_4^r) \uplus D_5^r \\
=& ((((((D_1 \bowtie (D_2 \sim D_3)) \cup (\delta_1 \bowtie (D_2 \sim D_3))) \cup ((D_4 \bowtie D_5) \sim \\
& D_6)) \uplus D_1^r) \uplus D_2^r) \uplus D_4^r) \uplus D_5^r \\
=& ((((((D_1 \bowtie (D_2 \sim D_3)) \cup ((D_4 \bowtie D_5) \sim D_6)) \cup (\delta_1 \bowtie (D_2 \sim \\
& D_3))) \uplus D_1^r) \uplus D_2^r) \uplus D_4^r) \uplus D_5^r \\
=& ((((f(D_1, \ldots, D_6) \cup (\delta_1 \bowtie (D_2 \sim D_3))) \uplus D_1^r) \uplus D_2^r) \uplus D_4^r) \uplus D_5^r
\end{aligned}$$

Using the same transformation procedure, a set of increment expressions for the rest of data containers is obtained as follows:

$$\begin{aligned}
\delta_1 :& ((((f(D_1, \ldots, D_6) \cup (\delta_1 \bowtie M_1)) \uplus D_1^r) \uplus D_2^r) \uplus D_4^r) \uplus D_5^r \\
\delta_2 :& ((((f(D_1, \ldots, D_6) \cup (D_1 \bowtie (\delta_2 \sim D_3))) \uplus D_1^r) \uplus D_2^r) \uplus D_4^r) \uplus D_5^r \\
\delta_3 :& ((((f(D_1, \ldots, D_6) \sim (\delta_3 \sim ((D_4 \bowtie D_5) \sim D_6))) \uplus D_1^r) \uplus D_2^r) \uplus D_4^r) \uplus D_5^r \\
=& ((((f(D_1, \ldots, D_6) \sim (\delta_3 \sim M_4)) \uplus D_1^r) \uplus D_2^r) \uplus D_4^r) \uplus D_5^r \\
\delta_4 :& ((((f(D_1, \ldots, D_6) \cup ((\delta_4 \bowtie D_5) \sim D_6)) \uplus D_1^r) \uplus D_2^r) \uplus D_4^r) \uplus D_5^r \\
\delta_5 :& ((((f(D_1, \ldots, D_6) \cup ((D_4 \bowtie \delta_5) \sim D_6)) \uplus D_1^r) \uplus D_2^r) \uplus D_4^r) \uplus D_5^r \\
\delta_6 :& ((((f(D_1, \ldots, D_6) \sim (\delta_6 \sim M_1))) \uplus D_1^r) \uplus D_2^r) \uplus D_4^r) \uplus D_5^r
\end{aligned}$$

where:

$$M_1 = ((D_2 \sim D_3) \uplus D_2^r)$$
$$M_2 = (((D_1 \bowtie (D_2 \sim D_3)) \uplus D_1^r) \uplus D_2^r)$$
$$M_3 = (((D_4 \bowtie D_5) \uplus D_4^r) \uplus D_5^r)$$
$$M_4 = ((((D_4 \bowtie D_5) \sim D_6) \uplus D_4^r) \uplus D_5^r)$$

Therefore:

$$g_1 = (\delta_1 \bowtie M_1)$$
$$g_2 = (D_1 \bowtie (\delta_2 \sim D_3))$$
$$g_3 = (\delta_3 \sim M_4)$$
$$g_4 = ((\delta_4 \bowtie D_5) \sim D_6)$$
$$g_5 = ((D_4 \bowtie \delta_5) \sim D_6)$$
$$g_6 = (\delta_6 \sim M_1)$$

□

If we remove the *minion* operations on *rover* data containers, increment expressions obtained are exactly the same as increment expressions discussed in Chapter 4.

6.5.3 Online Integration Plan for XML Fragments

Increment expressions generated from a data integration expression on fragmented XML documents are an extension of processing on complete XML documents. Therefore, the algorithms for online integration plans and scheduling described in the Chapter 4 dan be utilized. However, it is a freedom to use one of the following approaches to perform *minion* operations in an online integration plan:

1. All fragmented XML documents in the *rover* data containers are flushed whenever a fragmented XML document is sent to a *bounded* data container. To remove inconsistency between data containers and materializations, all fragmented XML documents in the *rover* data containers are combined to its corresponding data containers and materializations. The drawback of this approach is that *minion* operations need to be applied on all materializations.

   Example 6.12. *Generation of an online integration plan.*
   Let $g_1, \ldots, g_6$ be increment expressions generated in Example 6.11. Consider

a data increment (δ_2) which arrives at data container D_2. Transformation of an increment expression $g_2 = (D_1 \bowtie (\delta_2 \sim D_3))$ into an online integration plan d_2 is performed as follows:

(a) In the first step, an expression $(\delta_2 \sim D_3)$ is mapped into a step $\Delta_1 = (\delta_2 \sim D_3)$ and an expression $(D_1 \bowtie \Delta_1)$ into a step $\Delta_2 = (D_1 \bowtie \Delta_1)$;

(b) Then, $M_e = (M_e \sim \Delta_2)$ is appended to combine the computation results with the previous final materialization.

(c) $M_e = (M_e \uplus D_1^r); M_e = (M_e \uplus D_4^r); M_e = (M_e \uplus D_5^r)$ are appended to combine the *rover* data containers with the previous final materialization;

(d) The next step is a process to update a data container D_2: $D_2 = (D_2 \cup \delta_2)$;

(e) An intermediate materialization M_1 is identified to be affected to update. M_1 is result of computation of a data integration expression $h_1(D_2, D_3) = (D_2 \sim D_3)$, therefore $h_1(D_1, D_2)$ is transformed into an increment expression $g_{M1} = (\delta_2 \sim D_3)$, and a plan for updating M_1 is generated as follows: $d_{M1} : \Delta_{M1} = (\delta_2 \sim D_3); M_1 = (M_1 \cup \Delta_{M1}); M_1 = (M_1 \uplus D_3^r)$. These steps of processing are appended to the steps produced erlier.

Then, the complete online integration plan for increment expression g_2 is as follows:

$$
\begin{aligned}
&p_1 : \Delta_1 = (\delta_2 \sim D_3);\\
&p_2 : \Delta_2 = D_1 \bowtie \Delta_1;\\
&p_3 : M_e = (M_e \sim \Delta_3);\\
&p_4 : M_e = (M_e \uplus D_1^r);\\
&p_5 : M_e = (M_e \uplus D_4^r);\\
&p_6 : M_e = (M_e \uplus D_5^r);\\
&p_7 : D_2 = (D_2 \cup \delta_2);\\
&p_8 : \Delta_{M1} = (\delta_2 \sim D_3);\\
&p_9 : M_1 = (M_1 \cup \Delta_{M1})\\
&p_{10} : M_1 = (M_1 \uplus D_3^r);
\end{aligned}
$$

□

2. Since fragmented XML documents in the *rover* data containers have no effect on the rest of computation, *minion* operations are applied only at the very end of the computation process. In this approach, fragmented XML documentsin the *rover* data containers are excluded in computation of data integration expression until the results are ready to be sent to users.

   Example 6.13. *A second approach of an online integration plan.*
   Let $g_1, \ldots, g_6$ be increment expressions generated in Example 4.9. We consider a data increment (δ_2) arriving at a data container D_2. Transformation of an increment expression $g_2 = (D_1 \bowtie (\delta_2 \sim D_3))$ into an online integration plan d_2 is performed as follows:

   (a) In the first step, an expression $(\delta_2 \sim D_3)$ is mapped into a step $\Delta_1 = (\delta_2 \sim D_3)$ and an expression $(D_1 \bowtie \Delta_1)$ into a step $\Delta_2 = (D_1 \bowtie \Delta_1)$;

   (b) Then, $M_e = (M_e \sim \Delta_2)$ is appended to combine the computation results with the previous final materialization;

   (c) The next step is to update a data container D_2: $D_2 = (D_2 \cup \delta_2)$;

   (d) An intermediate materialization M_1 is identified to be affected to update. M_1 is a computation result of a data integration expression $h_1(D_2, D_3) = (D_2 \sim D_3)$, therefore $h_1(D_1, D_2)$ is transformed into an increment expression $g_{M1} = (\delta_2 \sim D_3)$. A plan to update M_1 is generated as follows: $d_{M1} : \Delta_{M1} = (\delta_2 \sim D_3); M_1 = (M_1 \cup \Delta_{M1})$. These steps of processing are appended to the steps produced erlier.

   Then, the complete online integration plan for increment expression g_2 is as follows:

$$
\begin{aligned}
&p_1 : \Delta_1 = (\delta_2 \sim D_3);\\
&p_2 : \Delta_2 = D_1 \bowtie \Delta_1;\\
&p_3 : M_e = (M_e \sim \Delta_3);\\
&p_4 : D_2 = (D_2 \cup \delta_2);\\
&p_5 : \Delta_{M1} = (\delta_2 \sim D_3);\\
&p_6 : M_1 = (M_1 \cup \Delta_{M1}).
\end{aligned}
$$

   As can be seen from the result, processing of an online integration plan for fragmented XML documents in this approach has the same algorithms as in Section 4.4.2.

Then, *minion* operations are performed to combine *rover* data containers with the final materialization before results are sent to users. The operations are performed as follows: $M_e = (M_e \uplus D_1^r); M_e = (M_e \uplus D_2^r); M_e = (M_e \uplus D_4^r); M_e = (M_e \uplus D_5^r)$. Operations to combine *rover* data containers to the intermediate materializations are not necessary since the end of computation process has been reached. □

To increase performance of the online integration system, the *defragmentation* procedure is applied at the very end of computation or only if needed. *Defragmentation* allows transformation of a fragmented XML document into an XML document.

At this stage, the algorithms on scheduling of online integration plans described in Section 4.5 are applicable.

In this chapter, it has been demonstrated that processing of fragmented XML documents improves the performance of the online data integration system by reducing the waiting time to trigger processing of a data increment. The minimum XML elements required to process a data increment have been identified to create an adequate list for the particular data container.

Some operations defined in the previous chapters have been modified, and new operators necessary for processing of XML fragments added.

Chapter 7

Summary, Conclusion and Future Works

A summary of this thesis is presented in Section 7.1 and contributions in Section 7.2. Then, the thesis is concluded in Section 7.3. Section 7.4 describes recommendations and some future works.

7.1 Summary

The approaches to data integration can be classified depending on the way in which data is accessed. It can be either a *materialization approach* or *virtualization approach*. In a *materialization approach*, a central site transmits and transforms data from the external sites and stores the results in a materialized copy which must be refreshed from time to time at a central site. A central site uses the materialized copies of data to process the queries. On the other hand, in a *virtualization approach*, a central site provides a virtual global view of the external data sources to a user, and no materialization of external data is performed at the central site. A query to a central site is translated into a number of sub-queries which are sent to the external sites for processing. Then, the central site combines the results and presents them to a user.

The data integration system described in this thesis is based on a *virtualization approach*. The data integration system takes a *user request* based on the virtual global view of external sites, and transforms it into a *global query expression*. Then, the mediator decomposes a *global query expression* to balance processing between the central site and remote sites. Then, a *data integration expression* is generated.

Online data processing allows for theoretically infinite sequences of input data to be processed to produce results. *Online data processing* employs *online algorithms* where input data are processed in a *piece-by-piece* mode without having the entire set of data available from the very beginning.

Online data integration is a process of continuous consolidation of data transmitted over a network with the data already available at the central site of a

distributed database environment. The system applies an *online algorithm* to process the units of data increment without having entire set of data available at the central site. An approach presented in this thesis transforms a data integration expression generated at an earlier stage into a number of *increment expressions* such that every argument of a data integration expression has an increment expression assigned to it. Then, a sequence of algebraic operations in an *online integration plan* is created for every increment expression.

Sequential processing of data increments which share intermediate materialization incurs an inefficient IO cost for loading it from and then storing it back to a persistent storage. Parallel processing of these increments is performed in order to optimize online integration plans by reducing the costs of operations on the same materialization. The increment expression is modified such that one increment expression covers processing of multiple data increments.

Since data increments arrive at the central site in random order, static scheduling of plans might create a poor performance of the system. In some cases we might want to wait until a number of data increments arrive before processing them in one step. The dynamic scheduling system proposed in this thesis optimizes execution of online integration plans by changing the order of data increment processing in a sliding window. Every increment in the sliding window is labeled with a priority label accordingly to their increment expression form (*union*-ed or *antijoin*-ed with previous result), their frequencies, and their types. Then, the order of increment processing is changed according to the priority labels of data increments. The scheduling system is able to ignore the remaining steps of a plan when an operation produces no result to pass to the next steps, as well as being able to update corresponding intermediate materializations if needed.

Processing of large size XML documents might reduce the performance of an online data integration because processing of an increment must wait until a complete XML document available at the central site. The online integration system is optimized by breaking down large size XML documents into XML fragments. Then, processing of a data increment is triggered when a fragmented XML document has enough properties to compute. To justify the sufficiency of increment's property, an adequate list is employed for every argument of a data integration expression.

The online data integration system presented in this thesis is designed to process semistructured data, where an XML document is used as a model for semistructured data. Semistructured data play an important role in a web data

interchange system since a relational database is too schema restricted and unstructured data is harder to analyze. Among semistructured data models, XML has advantages in name conflict resolution by using node indexing, and the ability to handle mixed contents. XML is supported by XPath, XML Schema and XML which set apart XML from JSON particularly. A different formalization of XML data model based on tree grammar is proposed. Then an XML algebra is proposed together with a set of algebraic operators which support online data integration system. A relational-like algebra is presented, where definition of tuples are replaced with XML documents which may have more complex structures. This contains a set of basic unary operators (*selection, projection*) and binary algebraic (*union,join,antijoin*) operators. The operators operate on multi-schema data containers of XML documents. Using the data containers with a set of schemas allows us to perform operations with less pain schema operations on participating XML documents.

7.2 Contributions

The first contribution is a new formal model of XML documents based on extended tree grammar and XML algebra. The XML algebra proposed in the thesis allows for online data integration.

The next contribution is an algorithm for integration of theoretically infinite sequences of semistructured data. This allows queries to be evaluated as soon as the units of data reach the central site. The algorithm has an online execution plan which allows parallel processing of data increments, and a dynamic scheduling system based on the data increments behavior.

The last contribution is an XML fragment model for the online data integration system. The thesis has proposed some additional XML algebraic operators such that the online integration algorithms are able to process data increments without having complete documents available at the central site. The thesis presents an algorithm to push processing of the remaining XML fragments at the very end, and provides a set of XML algebraic operators to compose the XML fragments into bigger chunks.

7.3 Conclusion

The XML data model and algebraic operators proposed in this thesis are complete enough for implementation of an online integration of semistructured data. The

union and *selection* operations are similar to the *union* and *filter* in TAX (Tree Algebra for XML). However, the *join* operator presented in the thesis preserves its consistency to *join* operator in the relational algebra instead of a *structural join* as in TAX.

We show that the XML algebra reduces to the relational algebra in the following circumstances: (1) XML documents have simple structures as tuples, and (2) every data container has a single set of schema. Furthermore, we show that the proposed XML algebraic operators are applicable for any XML document structures. The XML algebraic operators presented in the thesis are consistent with the relational algebraic operators. Consistency with the relational model allows for any existing optimization technique to be directly applied for data integration.

The query decomposition strategy proposed in this thesis provides balancing of the computations between the central site and the remote sites to ensure maximum resource utilization at both sites, and to reduce transmission costs of data transfer from remote sites to the central site.

Online integration algorithms optimizes the integration system by starting processing during data transmission to produce partial results earlier. Online processing reduces the data amount involved in a single XML algebraic operation, since data increments are relatively much smaller than those in a materialization. Partial results are instantly available to the user if the data integration expression has no decremental results (*antijoin* operations). Otherwise, correct answers are available to the user after data containers which cause decremental results are complete.

Online integration plan optimizes the process of integration by reducing IO costs to load and write materializations as all materialization update operations are placed at the end of a plan. To generate an increment expression for several data increments we effectively reuse increment expressions for smaller data increments without the transformation of a data integration expression from scratch.

The dynamic scheduling system proposed in the thesis provides a better performance by modifying the sequence of increments accordingly to their behaviors such that minimal IO costs to update materializations are required. When most data containers are empty (*initial state*) or most of the data containers are completed (*ending state*), it may happen that the execution of an online integration plan may not change the final result, and therefore is unnecessary. The dynamic scheduling system described in the thesis optimizes online integration plans through deferring or terminating a plan such that the total number of updates performed on the materializations is reduced.

The online integration system on fragmented XML documents adjusts the size of data increments to trigger their processing. It increases the performance of the integration system on large size documents because processing of an increment can be started earlier. In addition, processing of fragmented XML documents is able to stop unnecessary fragments to be processed, and therefore reduce computations. It is shown that the XML algebraic operations push processing of the remaining XML fragments at the end of a plan.

The pre-processing stage of the data integration approach proposed in the thesis requires a series of transformations from a data integration expression into a number of online integration plans. For a small amount of data, the pre-processing stage can be more expensive than any ordinary data integration system. Therefore, the system has better performance on processing large size and volume of semistructured data, such that the pre-processing cost is immaterial compared to the potential efficiencies of processors, transmission and IO.

The transformation of a *data integration expression* with n arguments into one *increment expression* is done through multiple applications of XML algebraic rules, and requires a linear time complexity $\mathcal{O}(n)$. An *online integration plan* can be generated from an *increment expression* in $\mathcal{O}(n)$. Whereas, the algorithm for *dynamic scheduling* includes labeling and sorting of data increments in a *sliding window*, and takes at most $\mathcal{O}(n \log n)$.

Finally, the thesis provides a clear formalization of the semistructured data model, a set of algorithms with high-level descriptions, and running examples. These formal backgrounds show that the algorithms proposed are implementable.

7.4 Recommendations and Future Work

First of all, the XML algebra can be expanded by adding XML algebraic operators with *aggregation* and *grouping* operations to power the online integration system with a data analysis feature.

Next, it has been found that online data integration system in this thesis is inefficient for integration of a small amount of semistructured data due to the high cost at the pre-processing phase to transform a user request until online integration plans are generated. If information about the size of data involved can be obtained earlier, a scheduling algorithm may ignore online processing for relatively small data and wait until all data are available at the central site before data integration computation is started.

The formal backgrounds and algorithms proposed are enough to allow an extension of this thesis with developed scenario and empirical studies. This opens future opportunities in implementation of the online integration system on resource-constrained devices whose limited processing and storage capabilities, and often runs on batteries.

In the thesis, sub-queries which resulting from the decomposition strategy are sent to the external site simultaneously, and remote sites perform the computation of the sub-queries in parallel fashion. To reduce data transmission, the sub-queries could be modified before they are sent to the remote sites according to any data available at the central site. For a number of sub-queries generated, some queries are sent to remote sites, and wait the results of these queries. Based on the results, the rest of the queries are modified before sending them to remote sites.

Online data integration described in this thesis is applicable for processing data streams, especially XML data streams and fragmented XML data streams. The stream data source is plugged into data containers as arguments of an online integration expression.

The internet of things (IoT) is interconnected physical devices, embedded systems, sensors and software which are able to exchange data. IoT is powered with a Wireless Sensor Network (WSN) which is interconnected sensors. WSN collects continuous information of sensors and broadcasts data to IoT. Information integration among objects in WSN requires *online processing* in order to get instant results of integration. The online integration system proposed in this thesis is applicable for integration data over IoT by replacing data containers with sensor nodes. The first challenge to be solved is the data structure alignment as sensors may have their own data structures. The second challenge is integration of numerous sensors. Then, the characteristic of data is a challenge where data size is usually small but frequently changes.

In a multi-database system where multiple sites act as "central site", we may want to modify the decomposition strategy such that the sub-queries generated can be sent either directly as queries to external sites where data are located or as data integration expressions to other "central site". The central site of a multi-database system no longer points to a dedicated site, but is transparent to the user. MapReduce allows processing of a data integration expression to be distributed into a number of data integration expressions and combines the results afterwards.

Bibliography

[1] Almarimi A. and Pokorný J. Schema management for data integration: A short survey. *Acta Polytechnica*, (1), 2005.

[2] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Incremental maintenance for materialized views over semistructured data. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 38–49, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[3] Susanne Albers and Stefano Leonardi. On-line algorithms. *ACM Comput. Surv.*, 31(3es), September 1999.

[4] L. Amsaleg, A. Tomasic, M. J. Franklin, and T. Urhan. Scrambling query plans to cope with unexpected delays. In *Parallel and Distributed Information Systems, 1996., Fourth International Conference on*, pages 208–219, Dec 1996.

[5] Yigal Arens, Craig A. Knoblock, and Wei-Min Shen. Query reformulation for dynamic information integration. *J. Intell. Inf. Syst.*, 6(2-3):99–130, August 1996.

[6] Apichaya Auvattanasombat, Yousuke Watanabe, and Haruo Yokota. XML documents searching combining structure and keywords similarities. *IPSJ SIG Technical Reports*, 2013(14):1–6, 2013.

[7] Catriel Beeri and Yariv Tzaban. SAL: An algebra for semistructured data and XML. In *Informal Proc. of Workshop on The Web and Databases, ACM SIGMOD*, pages 37–42. ACM Press, 1999.

[8] Leopoldo Bertossi and Loreto Bravo. Consistent query answers in virtual data integration systems. In Leopoldo Bertossi, Anthony Hunter, and Torsten Schaub, editors, *Inconsistency Tolerance*, pages 42–83. Springer-Verlag, Berlin, Heidelberg, 2004.

[9] L. Birhanu, S. Atnafu, and F. Getahun. Native XML document fragmentation model. In *Signal-Image Technology and Internet-Based Systems (SITIS), 2010 Sixth International Conference on*, pages 233–240, 2010.

[10] Angela Bonifati and Alfredo Cuzzocrea. *Efficient Fragmentation of Large XML Documents*, pages 539–550. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[11] Angela Bonifati, Martin Goodfellow, Ioana Manolescu, and Domenica Sileo. Algebraic incremental maintenance of XML views. *ACM Trans. Database Syst.*, 38(3):14:1–14:45, Sep 2013.

[12] Sujoe Bose and Leonidas Fegaras. Data Stream Management for Historical XML Data. *SIGMOD*, 99(3):403–422, 2004.

[13] Sujoe Bose, Leonidas Fegaras, David Levine, and Vamsi Chaluvadi. A query algebra for fragmented XML stream data. In *Proceeding of 9th International Conference on Data Base Programming Languages (DBPL)*, pages 275–277, Potsdam, Germany, September 6-8 2003.

[14] L. Bouganim, F. Fabret, C. Mohan, and P. Valduriez. Dynamic query scheduling in data integration systems. In *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 425–434, 2000.

[15] Luc Bouganim, Françoise Fabret, C Mohan, and Patrick Valduriez. A dynamic query processing architecture for data integration systems. *IEEE Data Eng. Bull.*, 23(2):42–48, 2000.

[16] Vanessa Braganholo and Marta Mattoso. A survey on XML fragmentation. *SIGMOD Rec.*, 43(3):24–35, December 2014.

[17] Giacomo Buratti. *A Model and an Algebra for Semi-Structured and Full-Text Queries*. PhD thesis, Informatica, Università di Bologna, Padova, 2007.

[18] Andrea Calì, Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. On the expressive power of data integration systems. In *Proceedings of the 21st International Conference on Conceptual Modeling*, ER '02, pages 338–350, London, UK, UK, 2002. Springer-Verlag.

[19] Nathalie Charbel, Joe Tekli, Richard Chbeir, and Gilbert Tekli. Resolving XML semantic ambiguity. In *EDBT*, pages 277–288, 2015.

[20] Dunren Che and Radiya M. Sojitrawala. DUMAX: a dual mode algebra for XML queries. In *Proceedings of the 2nd international conference on Scalable information systems*, InfoScale '07, pages 52:1–52:4, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[21] S. Chen, H. G. Li, J. Tatemura, W. P. Hsiung, D. Agrawal, and K. S. Candan. Scalable filtering of multiple generalized-tree-pattern queries over XML streams. *IEEE Transactions on Knowledge and Data Engineering*, 20(12):1627–1640, Dec 2008.

[22] Songting Chen, Hua-Gang Li, Junichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selçuk Candan. Twig2stack: Bottom-up processing of generalized-tree-pattern queries over XML documents. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 283–294. VLDB Endowment, 2006.

[23] Zhimin Chen, H. V. Jagadish, Laks V. S. Lakshmanan, and Stelios Paparizos. From tree patterns to generalized tree patterns: On efficient evaluation of XQuery. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 237–248. VLDB Endowment, 2003.

[24] Vassilis Christophides, Sophie Cluet, and Jérŏme Simèon. On wrapping query languages and efficient XML integration. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 141–152, New York, NY, USA, 2000. ACM.

[25] An Hai Doan, Alon Halevy, and Zachary Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012.

[26] Xin Luna Dong, Alon Halevy, and Cong Yu. Data integration with uncertainty. *The VLDB Journal*, 18(2):469–500, 2009.

[27] D. Draper, A.Y. Halevy, and D.S. Weld. The Nimble XML data integration system. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 155–160, 2001.

[28] Denise Draper, Alon Y. Halevy, and Daniel S. Weld. The Nimble integration engine. In *Proceedings of the 2001 ACM SIGMOD International Conference*

on Management of Data, SIGMOD '01, pages 567–568, New York, NY, USA, 2001. ACM.

[29] Maged EL-Sayed, Ling Wang, Luping Ding, and Elke A. Rundensteiner. An algebraic approach for incremental maintenance of materialized XQuery views. In *Proceedings of the 4th International Workshop on Web Information and Data Management*, WIDM '02, pages 88–91, New York, NY, USA, 2002. ACM.

[30] Leonidas Fegaras. Incremental maintenance of materialized XML views. In Abdelkader Hameurlain, StephenW. Liddle, Klaus-Dieter Schewe, and Xiaofang Zhou, editors, *Database and Expert Systems Applications*, volume 6861 of *Lecture Notes in Computer Science*, pages 17–32. Springer Berlin Heidelberg, 2011.

[31] Leonidas Fegaras. Incremental query processing on Big Data streams. *CoRR*, abs/1511.07846, 2015.

[32] Leonidas Fegaras, David Levine, Sujoe Bose, and Vamsi Chaluvadi. Query processing of streamed XML data. In *Proceedings of the Eleventh International Conference on Information and Knowledge Management*, CIKM '02, pages 126–133, New York, NY, USA, 2002. ACM.

[33] Jamel Feki, Ines Ben Messaoud, and Gilles Zurfluh. Building an XML document warehouse. *Journal of Decision Systems*, 22(2):122–148, 2013.

[34] Mary Fernández, Jérôme Siméon, Byron Choi, Amélie Marian, and Gargi Sur. Implementing XQuery 1.0: The Galax experience. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 1077–1080. VLDB Endowment, 2003.

[35] Amos Fiat. Online algorithms: The state of the art (lecture notes in computer science). 1998.

[36] Ian Foster and Robert L. Grossman. Data integration in a bandwidth-rich world. *Commun. ACM*, 46(11):50–57, Nov 2003.

[37] Flavius Frasincar, Geert-Jan Houben, and Cristian Pau. XAL: an algebra for XML query optimization. *Australian Computer Science Communication*, 24(2):49–56, January 2002.

[38] Pierre Geneves, Nabil Layaïda, and Vincent Quint. Impact of XML schema evolution. *ACM Transactions on Internet Technology (TOIT)*, 11(1):4, 2011.

[39] Janusz R. Getta. Optimization of task processing schedules in distributed information systems. In *Informatics Engineering and Information Science*, volume 253 of *Communications in Computer and Information Science*, pages 333–345. Springer Berlin Heidelberg, 2011.

[40] J.R. Getta. Query scrambling in distributed multidatabase systems. In *Database and Expert Systems Applications, 2000. Proceedings. 11th International Workshop on*, pages 647–652, 2000.

[41] J.R. Getta. Optimization of online data integration. In *Proceeding of 2006 Seventh International Baltic Conference on Databases and Information Systems*, pages 91–97, 2006.

[42] Anastasios Gounaris, Norman W. Paton, Alvaro A.A. Fernandes, and Rizos Sakellariou. Self-monitoring query execution for adaptive query processing. *Data & Knowledge Engineering*, 51(3):325 – 348, 2004.

[43] Timothy Griffin, Leonid Libkin, and Howard Trickey. An improved algorithm for the incremental recomputation of active relational expressions. *IEEE Trans. on Knowl. and Data Eng.*, 9(3):508–511, May 1997.

[44] Paolo Guagliardo and Piotr Wieczorek. Query processing in data integration. *Dagstuhl Follow-Ups*, 5, 2013.

[45] M. Hachicha and J. Darmont. A survey of XML tree patterns. *Knowledge and Data Engineering, IEEE Transactions on*, 25(1):29–46, 2013.

[46] Alon Halevy, Anand Rajaraman, and Joann Ordille. Data integration: The teenage years. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 9–16. VLDB Endowment, 2006.

[47] Alon Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, December 2001.

[48] Handoko and Janusz R. Getta. An XML algebra for online processing of XML documents. In *The 15th International Conference on Information Integration and Web-based Applications & Services, IIWAS '13, Vienna - December 2-4*, 2013.

[49] Handoko and Janusz R. Getta. Dynamic query scheduling for online integration of semistructured data. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, volume 3, pages 375–380, July 2015.

[50] Jorng-Tzong Horng, Yu-Jan Chang, Baw-Jhiune Liu, and Cheng-Yan Kao. Materialized view selection using genetic algorithms in a data warehouse system. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 3, page 2227 Vol. 3, 1999.

[51] Haruo Hosoya. *FOUNDATIONS OF XML PROCESSING - The Tree-Automata Approach*. Cambridge University Press, United Kingdom, 2011.

[52] Huo Huan, Wang Guoren, Chen Qingkui, and Peng Dunlu. SLCA algorithm for XML streams based on hole-filler model. *Journal of Computer Research and Development*, 47(5):886, 2010.

[53] Huan Huo, Qingkui Chen, Guoren Wang, Dunlu Peng, Jutao Hao, and Liping Gao. The cost model for fragmented XML streams. In *Information Science and Engineering (ICISE), 2010 2nd International Conference on*, pages 2428–2431, 2010.

[54] Huan Huo, Guoren Wang, Xiaoyun Hui, Rui Zhou, Bo Ning, and Chuan Xiao. *Efficient Query Processing for Streamed XML Fragments*, pages 468–482. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[55] Zachary G. Ives. *Efficient Query Processing for Data Integration*. PhD thesis, Department of Computer Science and Engineering, USA, 2002.

[56] Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Daniel S. Weld. An adaptive query execution system for data integration. *SIGMOD Rec.*, 28(2):299–310, June 1999.

[57] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A tree algebra for XML. In *In Proc. DBPL Conf*, pages 149–164, 2001.

[58] Prudhvi Janga and Karen C Davis. Schema extraction and integration of heterogeneous XML document collections. In *International Conference on Model and Data Engineering*, pages 176–187. Springer, 2013.

[59] Kim Jin, Kang Hyunchul, and Jacek Kitowski. A method of XML document fragmentation for reducing time of XML fragment stream query processing. *Computing & Informatics*, 31(3):639 – 664, 2012.

[60] T. Jörg. *Incremental Recomputations in Materialized Data Integration*. PhD thesis, Fachbereich Informatik - Technischen Universität Kaiserslautern, 2013.

[61] Thomas Jörg, Roya Parvizi, Hu Yong, and Stefan Dessloch. Incremental recomputations in mapreduce. In *Proceedings of the Third International Workshop on Cloud Data Management*, CloudDB '11, pages 7–14, New York, NY, USA, 2011. ACM.

[62] Vanja Josifovski, Marcus Fontoura, and Attila Barta. Querying XML streams. *The VLDB Journal*, 14(2):197–210, 2005.

[63] Christoph Koch and Stefanie Scherzinger. *Attribute Grammars for Scalable Query Processing on XML Streams*, pages 233–256. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[64] Laks S. V. Lakshmanan and M. TAMER ÖZSU. *XML Tree Pattern, XML Twig Query*, pages 3637–3640. Springer US, Boston, MA, 2009.

[65] Andreas Langegger. Virtual data integration on the web: Novel methods for accessing heterogeneous and distributed data with rich semantics. In *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '08, pages 559–562, New York, NY, USA, 2008. ACM.

[66] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 233–246, New York, NY, USA, 2002. ACM.

[67] Alon Y. Levy. Logic-based artificial intelligence. chapter Logic-based Techniques in Data Integration, pages 575–595. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[68] Jian Liu and X.X. Zhang. Data integration in fuzzy XML documents. *Information Sciences*, 280(0):82 – 97, 2014.

[69] Bertram Ludäscher, Yannis Papakonstantinou, and Pavel Velikhov. *Navigation-Driven Evaluation of Virtual Mediated Views*, pages 150–165. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[70] Maxim Lukichev and Dmitry Barashev. XML query algera for cost-based optimization. In *Proceedings of the SYRCoDIS 2007 Colloquium on Databases and Information Systems, Moscow, Russia, May 31 - June 1*, 2007.

[71] Maxim Lukichev, Boris Novikov, and Pankaj Mehra. An XML-algebra for efficient set-at-a-time execution. *ComSIS*, 9(1):64–80, January 2012.

[72] Hui Ma and Klaus-Dieter Schewe. Fragmentation of XML documents. *Journal of Information and Data Management*, 1(1):21 – 33, 2010.

[73] Ashok Malhotra, Kristoffer Rose, Peter Fankhauser, Mary Fernandez, Michael Dyck, Philip Wadler, Michael Rys, Jerome Simeon, and Denise Draper. XQuery 1.0 and XPath 2.0 formal semantics (second edition). W3C recommendation, W3C, December 2010. http://www.w3.org/TR/2010/REC-xquery-semantics-20101214/.

[74] Dimitris Manakanatas and Dimitris Plexousakis. A tool for semi-automated semantic schema mapping: Design and implementation. In *DISWEB*. Citeseer, 2006.

[75] Murali Mani and Dongwon Lee. *XML to Relational Conversion Using Theory of Regular Tree Grammars*, pages 81–103. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[76] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Answering XML queries on heterogeneous data sources. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 241–250, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[77] Wolfgang May. Logic-based XML data integration: a semi-materializing approach. *Journal of Applied Logic*, 3(2):271 – 307, 2005.

[78] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Technol.*, 5(4):660–704, November 2005.

[79] Tamer M. Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, third edition*. Springer, 2011.

[80] Stelios Paparizos, Shurug Al-Khalifa, H.V. Jagadish, Andrew Nierman, and Yuqing Wu. A physical algebra for XML. [http://dbgroup.eecs.umich.edu/timber/files/physical.pdf, accessed 15-June-2016].

[81] Stelios Paparizos and H. V. Jagadish. Pattern tree algebras: Sets or sequences? In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 349–360. VLDB Endowment, 2005.

[82] Stelios Paparizos and H. V. Jagadish. *The Importance of Algebra for XML Query Processing*, pages 126–135. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[83] Stelios Paparizos, Yuqing Wu, Laks V. S. Lakshmanan, and H. V. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 71–82, New York, NY, USA, 2004. ACM.

[84] E. Peukert, J. Eberius, and E. Rahm. A self-configuring schema matching system. In *2012 IEEE 28th International Conference on Data Engineering*, pages 306–317, April 2012.

[85] Maciej Piernik, Dariusz Brzezinski, Tadeusz Morzy, and Anna Lesniewska. XML clustering: a review of structural approaches. *The Knowledge Engineering Review*, 30(03):297–323, 2015.

[86] Antonella Poggi and Serge Abiteboul. XML data integration with identification. In *Database Programming Languages*, volume 3774 of *Lecture Notes in Computer Science*, pages 106–121. Springer Berlin Heidelberg, 2005.

[87] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, December 2001.

[88] Vijayshankar Raman and Joseph M. Hellerstein. Partial results for online query processing. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 275–286, New York, NY, USA, 2002. ACM.

[89] Patricia RodrÌguez-Gianolli and John Mylopoulos. *A Semantic Approach to XML-based Data Integration*, pages 117–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[90] Deise Saccol and Carlos Heuser. Integration of XML data. In *Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web*, volume 2590 of *Lecture Notes in Computer Science*, pages 68–80. Springer Berlin Heidelberg, 2003.

[91] Rashed Salem, Omar Boussaid, and Darmont. Active XML-based Web data integration. *Information Systems Frontiers*, 15(3):371–398, 2013.

[92] Lee Sangwook, Kim Jin, and Kang Hyunchul. Memory-efficient query processing over XML fragment stream with fragment labeling. *Computing & Informatics*, 29(4):757 – 782, 2010.

[93] Carlo Sartiani and Antonio Albano. Yet another query algebra for XML data. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS'02)*, Edmonton, Canada, July 17-19 2002.

[94] V. Srinivasan and Michael J. Carey. Compensation-based on-line query processing. In *In Proc. ACMSIGMOD International Conference on Management of Data*, pages 331–340, 1992.

[95] N. Take, M. Nishio, and H. Seshake. Oss data integration using virtual database. In *Network Operations and Management Symposium (APNOMS), 2013 15th Asia-Pacific*, pages 1–6, Sept 2013.

[96] Nan Tang, Jeffrey Xu Yu, M Tamer Ozsu, Byron Choi, and Kam-Fai Wong. Multiple materialized view selection for xpath query rewriting. In *2008 IEEE 24th International Conference on Data Engineering*, pages 873–882. IEEE, 2008.

[97] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost-based query scrambling for initial delays. *SIGMOD Rec.*, 27(2):130–141, June 1998.

[98] Waraporn Viyanon, Sanjay K. Madria, and Sourav S. Bhowmick. XML data integration based on content and structure similarity using keys. In *On the Move to Meaningful Internet Systems: OTM 2008*, volume 5331 of *Lecture Notes in Computer Science*, pages 484–493. Springer Berlin Heidelberg, 2008.

[99] Guoren Wang, Huan Huo, Donghong Han, and Xiaoyun Hui. Query processing and optimization techniques over streamed fragmented XML. *World Wide Web*, 11(3):339–359, 2008.

[100] Wikipedia. Online algorithm, July 2013.

[101] Xiaoying Wu and Dimitri Theodoratos. A survey on XML streaming evaluation techniques. *The VLDB Journal*, 22(2):177–202, 2013.

[102] Xin Wu and Guiquan Liu. XML twig pattern matching using version tree. *Data & Knowledge Engineering*, 64(3):580 – 599, 2008.

[103] Xiao-Quan Yan and Yuan Liu. XQuery optimization in heterogeneous data integration system. In *Management and Service Science, 2009. MASS '09. International Conference on*, pages 1–6, Sept 2009.

[104] Jian Yang, Kamalakar Karlapalem, and Qing Li. Algorithms for materialized view design in data warehousing environment. In *VLDB*, volume 97, pages 136–145, 1997.

[105] Liang Huai Yang, Mong Li Lee, and Wynne Hsu. Efficient mining of XML query patterns for caching. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 69–80. VLDB Endowment, 2003.

[106] Chen Jason Zhang, Ziyuan Zhao, Lei Chen, H. V. Jagadish, and Chen Caleb Cao. Crowdmatcher: Crowd-assisted schema matching. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 721–724, New York, NY, USA, 2014. ACM.

[107] Xin Zhang, Bradford Pielech, and Elke A. Rundesnteiner. Honey, I shrunk the XQuery!: an XML algebra optimization approach. In *Proceedings of the 4th international workshop on Web information and data management*, WIDM'02, pages 15–22, New York, NY, USA, 2002. ACM.

[108] Xin Zhang and Elke A. Rundensteiner. XAT: XML algebra for the rainbow system., 2002. [http://dbgroup.eecs.umich.edu/timber/files/physical.pdf, accessed 1-June-2016].