The XFM View Adaptation Mechanism: An Essential Component for XML Data Warehouses

Jun Liu

Bachelor of Science in Computer Science

A Dissertation submitted in fulfilment of the

requirements for the award of

Doctor of Philosophy (Ph.D.)

to the

Dublin City University

Faculty of Engineering and Computing, School of Computing

Supervisor: Mark Roantree

August 2011

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy is entirely my own work, that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: _____ ID No.: 57115001 Date: August 24, 2011

Acknowledgements

I would like to thank all those people who made this dissertation possible. In particular, I wish to express my sincere gratitude to my supervisor Dr. Mark Roantree for his patience guide, effort, encouragement and excellent advices throughout the PhD project. Without Mark, this dissertation would not have been possible.

A special note of thanks goes to Prof. Zohra Bellahsene for her guidance and support on my initial entry into the project.

Thanks also to Enterprise Ireland who supplied the funding for my research and to Dublin City University for the various structures and support they provided.

Thanks to my colleagues from the Interoperable Systems Group for sharing their experiences and knowledge during the time of my study. Especially to Martin and Gerard for helping me improve my writing skill and thrashing out various idea.

Finally, I would like to express my deepest gratitude to my wife Fangfang for all her support and encouragement during my PhD study.

Contents

List of Tables

List of Figures

List of Algorithms

Abstract

In the past few years, with many organisations providing web services for business and communication purposes, large volumes of XML transactions take place on a daily basis. In many cases, organisations maintain these transactions in their native XML format due to its flexibility for exchanging data between heterogeneous systems. This XML data provides an important resource for decision support systems. As a consequence, XML technology has slowly been included within decision support systems of data warehouse systems. The problem encountered is that existing native XML database systems suffer from poor performance in terms of managing data volume and response time for complex analytical queries. Although materialised XML views can be used to improve the performance for XML data warehouses, update problems then become the bottleneck of using materialised views. Specifically, synchronising materialised views in the face of changing view definitions, remains a significant issue. In this dissertation, we provide a method for XML-based data warehouses to manage updates caused by the change of view definitions (*view redefinitions*), which is referred to as the *view adaptation* problem. In our approach, views are defined using XPath and then modelled using a set of novel algebraic operators and fragments. XPath views are integrated into a single view graph called the XML Fragment Materialisation (XFM) View Graph, where common parts between different views are shared and appear only once in the graph. Fragments within the view graph can be selected for materialisation to facilitate the view adaptation process. While changes are applied, our view adaptation algorithms can quickly determine what part of the XFM view graph is affected. The adaptation algorithms then perform a *structural adaptation* to update the view graph, followed by *data adaptation* to update materialised fragments.

Chapter 1

Introduction

The eXtensible Markup Language (XML) has become an extraordinarily popular format for marking up all kinds of data, e.g., web content, sensor data and data used in many online applications. Due to its flexibility, it has widespread usage in terms of data storage, exchange and display. As a consequence, there are compelling reasons for using XML and related database technologies for querying and manipulating XML data. However, its flexibility and rich semantics have a downside: queries across large XML repositories are often slow. In this chapter, we provide an overview of XML and related technologies, describe a popular approach to optimisation, and by highlighting the limitations to this approach, provide a motivation and work plan for the research presented in this thesis.

1.1 The Emergence of XML

In the 1970's, three researchers at IBM invented GML, to provide a means of marking up technical documents with structural tags. GML developed into the Standard Generalised Markup Language (SGML) and was adopted by the ISO in 1986. SGML is a specification for defining markup languages rather than a markup language itself. A well known application of SGML is HTML (Hypertext Markup Language), which defines a specific set of tags suitable for web pages.

HTML became the means for separating content from presentation so that web browsers can render them in as consistent a manner as possible. However, when it comes to data storage and exchange, HTML is not a best choice, as it was originally designed as a presentation

technology. Furthermore, SGML is not suited to exchanging information over the web and is considered too complex for general usage. As a result, the eXtensible Markup Language (XML) was created to bridge this gap as its characteristics made it flexible enough to be a support platform and architecture independent data exchange mechanism so that richly structured data could be exchanged over the web and other applications.

In parallel to these developments, e-commerce has seen a tremendous amount of business transactions being conducted over the Internet. It is believed that XML is one of the best vehicles for exchanging business transactions on the Internet [LWC06] due to the powerful capabilities brought by XML [Bra03] as listed below:

- Heterogeneity: Where each "record" can contain different data fields. The real world is not neatly organised into tables, rows, and columns. There is great advantage in being able to express information, as it exists, without restrictions.
- Extensibility: Where new types of data can be added as needed and do not need to be determined in advance.
- Flexibility: Where data fields can vary in size and configuration from instance to instance. XML imposes no restriction on data; each data element can be as long or as short as necessary.

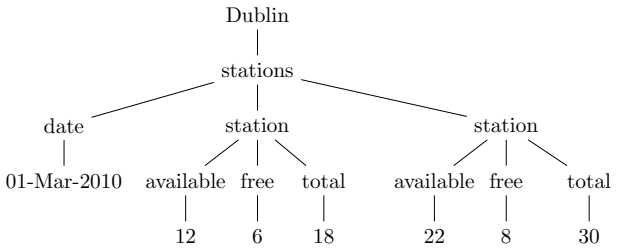
Due to its applicability, we now see an increasingly large amount of business transactions has been stored in XML format and exchanged online. Additionally, XML is also an essential tool for building *digital government* which refers to the possibilities to utilise current and future information and communication technologies (ICT) effectively to build new kinds of services both for people working in public sectors and for people needing their services [Sal05].

The exploitation of XML for e-commerce and government is also matched by new forms of web services and in recent times, the sensor web [RS09]. Large volumes of XML data are likely to be generated in Sensor Web systems due to its highly interoperable properties, which is crucial when integrating sensor data, both within a single sensor network and with the data generated by other sensor networks [RS09]. One such example demonstrates how raw data output from sensor devices is structurally and semantically enriched using XML

```
<Dublin>
    <stations date="01-Mar-2010">
        <station>
            <available>12</available>
            <free>6</free>
            <total>18</total>
        </station>
        <station>
            <available>22</available>
            <free>8</free>
            <total>30</total>
        </student>
    </stations>
</Dublin>
```

(a) XML Document

(c) XPath Query : //Dublin//station[.//total>20]



(b) XML Tree

Figure 1.1: Sample XML Document and XML Tree

for heart rate monitors worn by players in team sports [MRMW09]. What started out as an emerging model for information enrichment and exchange, has now become so common place and used in diverse scenarios and applications, that we now see considerable volumes of both XML transactions and repositories.

1.2 XML and Structure

Before proceeding to describe both structure and storage for XML data, we now briefly introduce the dataset that will be used throughout the thesis for motivating issues and presenting examples. Figure 1.1 is a sample segment of the Worldbikes dataset, which is a collection of sensor data that are continuously collected from the Bicycle Sharing System over the world. As discussed in Chapter 4, the collection is through a daily based automatic process with data obtained from 635 stations of 7 different cities over the world, e.g., Dublin, Lyon and Brisbane.

Figure 1.1a is a segment of the XML document of Worldbikes data which contains structured information. The content of the dataset is encapsulated with elements that are defined by tags, e.g., *stations* in Figure 1.1a. Those tags are nested and self-descriptive, which help users or applications understand the content of XML documents. Elements within the XML dataset are hierarchically organised into a tree-based structure as shown in Figure 1.1b. Additional descriptive information, namely attributes, may be included in tags. XML content can be accessed with different languages, XPath [W3C10a] and XQuery [W3C10b]. XPath is used to navigate and select nodes from XML documents. It is designed to be embed-

ded in a host language such as XQuery. Therefore, many research efforts focus on XPath. Figure 1.1c gives an example of an XPath expression which retrieves all *stations* in *Dublin* that have more than *20* bikes in total. In the example, *Dublin*, *stations* and *total* are referred to as the *NameTest*s in XPath, whereas the double slash (//) and single slash (/) represent either ancestor-descendant or parent-child relationships between NameTests. The ancestor-descendant and parent-child relationships are known as the *descendant* and *child axis*, respectively, in the XPath language.

XML datasets are semi-structured, which means no schema is required with data self-describing. However, XML documents can be associated with and validated against either a Document Type Definition (DTD) file [W3C08] or an XML Schema file [W3C04]. Both DTD and XML Schema describe the structure of XML documents and express constraints about the contents of XML documents. In order to obtain a better understanding of how these query languages and schemas are used, we now discuss storage mechanisms for XML data repositories.

1.2.1 XML Databases

Currently there are two types of XML databases: XML-Enabled Databases and Native XML Databases. The XML-enabled databases are built on top of relational databases and apply relational techniques over the XML data, e.g., traditional join algorithms like merge-join and hash-join used in the relational database systems. XML documents are converted to the relational model and stored in tables. The major problems as stated in [LWC06] with the XML-enabled databases are:

- As XML is expressed in tree structure, data must be converted into relational data format, a model with lesser semantics.
- Parts of the XML data structures, such as Prolog Instructions and comments are lost after converting XML documents into relational database systems. As a consequence, it is hard to restore an XML document with its original structure from the rational database systems.
- XML elements can be arbitrarily inserted into an XML document due to its heterogeneity. However, relational database systems require that all schemas are predefined

before any value can be inserted.

- Relational database systems must transform XML queries into the corresponding SQL queries.

For the above reasons, native XML databases emerged. As defined by the XML DB consortium, the formal definition of a Native XML Database states that it must have the following properties:

- Defines a (logical) model for an XML document;
- Has an XML document as its fundamental unit of (logical) storage, just as a relational database has a row in a table as its fundamental unit of (logical) storage;
- Need not have any particular underlying physical storage model. For example, Native XML Databases can use relational, hierarchical, or object-oriented database structures, or use a proprietary storage format (such as indexed, compressed files).

Native XML databases utilise a set of XML-specific technologies for representing and querying XML data. Within native XML databases, data objects are represented by using labelling schemes. Each node within an XML tree is assigned a label which uniquely identifies it. Additionally, XML-specific algorithms are used for processing XML data, e.g., twig join algorithms [BKS02, QYD07, LR10]. The XML-specific algorithms are performed upon an *encoding scheme*, which is constructed based on a labelling scheme and augments it with the information necessary to perform query processing [OR10]. As mentioned above, the physical storage of how those labelling schemes or encoding schemes are stored is irrelevant as long as XML-specific technologies are used. Examples of native XML databases are eXist [eXist], MonetDB [MonetDB] and BaseX [BaseX].

1.2.2 XML Data Warehouses

Since the Internet has evolved into a global platform for e-commerce and information exchange, the interest in XML has been growing significantly and large volumes of XML data already exist to provide an important resource for decision support systems [GRV01]. Furthermore, as more organisations view the web as integral to their communication and

business strategies, the importance of integrating XML data in data warehousing systems is becoming increasingly high. As a consequence, XML technology has slowly been included into the decision support process, e.g., data warehouses.

Existing efforts on integrating XML data into traditional data warehouses [NNNT02, RRT04], shows data being extracted from the XML documents and converted to the native format of the data warehouse (usually a relational database). An issue with this approach is the conversion of the heterogeneous data structure that is implied by the XML structure into a unified relational database schema. In a second approach, XML documents are either logically integrated within an XML warehouse by storing XML documents apart from the warehouse [PRP02, PP03, PPP04], or physically integrated within the warehouse by storing documents within data warehouse operating on native XML databases [ZWLZ05, NDRR06]. An issue with this approach is the poor performance when retrieving XML data due to the lack of maturity of XML databases. Other efforts merely differ from each other based on how they enable decision-support process, e.g., On-line Analytical Processing (OLAP), while an XML-based data warehouse is used. What all of this shows, is that the worlds of XML and data warehousing are beginning to converge.

1.3 View Materialisation and Adaptation

As this dissertation is based on XML data warehouses that use materialised views to boost query performance, it is necessary to provide a brief overview of the process and associated issues. As stated in [RTTZ10], existing XML-based data warehouse techniques all suffer from performance issues when using native XML databases. The reason is that native XML databases do not perform well when dealing with large data volumes and complex analytical queries that are typical in data warehousing [MD09]. Moreover, as XPath is a navigational language, XPath expressions often define complicated navigation over XML trees, which result in expensive query processing, especially when queries are executed over a large set of XML data. As a consequence, an extensive amount of research has been carried out in the past decade using materialised views [BOB$^+$04, LWZ06, TYÖ$^+$08a, WTW09, WLY11] to expedite XML query performance. Defining a small set of materialised views may result in avoiding complex computations and thus, yield important performance improvements for a

large set of queries [ABMP07].

Although using materialised views improves query performance, this approach suffers from an *inconsistency* problem, caused by either an update of the underlying databases or changes to view definitions. Existing efforts focus on providing *view maintenance* mechanisms [STP+05, OCMH05, JL10] which synchronise materialised views as the underlying data source is updated. There has been little research activity focused on developing approaches to keep views updated in response to view definition changes (or *view redefinition*). The process that handles view redefinition is called *view adaptation* and was first outlined by Gupta [GMR95] in relational database systems. The main objective of view adaptation is to *adapt* existing materialised data in response to view definition changes so that view definitions and associated materialised data, remain consistent.

Generally speaking, there are two methods to manage view redefinition. When a view is redefined, if the new view resulting from the redefinition is obtained by utilising previously materialised data, then this process is called *adapting* (*incremental adaptation of*) the view. The adaptation process usually adds (removes) data into (from) previously materialised data. However, when the result of the new view is obtained by evaluating the new view definition from scratch, the process is then referred to as *recomputing* or *rematerialising* the view.

1.4 Issues and Motivation

Materialised views have been shown to offer significant performance gains across different forms of data warehousing technologies. However, a crucial feature of these systems is the view adaptation component, that enables consistency between view definitions and materialised data. However, the primary problem is that current view adaptation methods for XML data warehouses have significant shortcomings, as we will point out in this section.

In relational database systems, the view adaptation problem has been studied in the context of both centralised [GMR95, GMRR01] and distributed environments [MD96, Moh97, Bel98, Bel00, Bel04]. In a centralised environment, [GMR95] provides a single-view based adaptation approach which augments views by adding extra attributes, mainly foreign keys. The effectiveness of the adaptation relies heavily on the existence of these foreign keys. In

their approach, they did not take network communication costs into account which is essential in the data warehousing environment. [MD96] improved the view adaptation approach presented in [GMR95] by augmenting the base relations and views with "join-count" and "derive-count" attributes. They analyse the network communication between different sites, e.g, they assume views and source data are distributed at different locations over the Internet. [Moh97] refines the work in [MD96] by providing new forms of view changes. Their approach is based on *expression trees* that consist of relational algebraic operators where changes can be made to the algebraic operators. However, all adopt a *single view* based approach and view adaptation algorithms are limited to the amount of materialised data that can be reused for adaptation. For this reason, the research in [Bel00, Bel04] provides a view adaptation approach based on a multi-view framework. In their approach, materialised data are shared between views by means of fragments. Fragments can be selected for materialisation provided that a selection algorithm is utilised. If any change is made to the view definition, their approach first checks to see whether the local shared materialised data can be reused so that access to the remote data source can be avoided. However, due to the difference between structured relational data, which is flat, regular, homogeneous and unordered, and semi-structured XML data, which is nested, irregular, heterogeneous and ordered, existing relational approaches cannot be directly adopted by XML warehouse systems.

The view adaptation problem provides very different challenges due to the particular characteristics of XML. To the best of our knowledge, [AML$^+$07] is the sole research effort focusing on view adaptation for XML database systems. In their approach, views are represented by a combination of union and intersection of XPath expressions. However, we will show that their approach handles very limited changes, either adding or removing an XPath expression from the view representation, and they do not take network communication cost (as studied in [MD96]) and the shareability of the materialised data (as studied in [Bel04]) into account.

In summary, existing XML approaches to view materialisation adopt an approach where a view is materialised as a single entity, for each view definition. To understand the issues inherent in this approach, let us look at the advantages of a fragmented approach as proposed in [Bel04]. A fragmented approach has the following benefits: materialised data

is shared which avoids duplication; the view adaptation performance is improved as there is a greater chance that materialised data can be reused; and finally, the number of access operations to source data is decreased, which reduces the network communication cost and further improves the view adaptation performance. This is important for distributed data warehouse systems. While all of these benefits are enjoyed by a fragmented approach to view materialisation, the sole fragmented approach [Bel04] operates only on relational data.

1.4.1 Research Goals and Contribution

Given the issues described in the previous section, we can now present the research goals that form the workplan for this dissertation. Our overall research goal is to provide query optimisation for XML using a materialisation approach. Having highlighted the issues when views are disjoint entities, materialised in full, our approach is to create a network of smaller materialised partitions. The hypothesis put forward in this research is that the view adaptation component, crucial to view maintenance, is more efficient when using a multi-view based framework with materialised data shared between views. The research goals necessary to deliver XML view adaptation can now be listed:

- We must define an XML view model with sufficiently expressive constructs and algebraic operators, to represent XML views.
- It is then necessary to develop a process whereby the XML view is transformed into our view model.
- To develop the view adaptation process which updates the view graph after applying changes to view definitions. This includes containment checking algorithms, which form an essential component of view adaptation. Its purpose is in identifying common expressions between views and determining the extent of changes between old view definitions and the new ones.
- To optimise the view adaptation process and to balance the cost of query performance and view maintenance, it is necessary to develop a process which selects the best view fragments within the view model for materialisation.

- Once we have components for both containment checking and selection, our view adaptation process is complete. However, it is essential that a framework is devised to enable each of the different components to interact.

The overall contribution of this dissertation is to provide a new view adaptation system for XML databases and warehouses. As part of this research, it was necessary to deliver:

- An XML view model consisting of a set of novel algebraic operators and view constructs.
- A novel transformation process which transforms XML views into our view model.
- A containment checking algorithm that exploits a novel XML metadata construct to improve speed and accuracy.
- A novel cost-based fragment selection algorithm.

1.5 Summary and Dissertation Structure

In this chapter, a general introduction to the eXtensible Markup Language, XML databases and XML-based data warehouses was provided. Due to the powerful features of heterogeneity, extensibility and flexibility brought by XML, it has been widely adopted as an exchange format for heterogeneous data sources from any forms of information systems. The interest in XML has been growing significantly and large volumes of XML data already exist which provide an important resource for decision support systems. More and more users of XML applications are seeking for solutions to integrate XML data into their decision support systems.

A major obstacle for XML-based data warehousing systems is the performance issue of the native XML databases. Although materialised XML views are used to expedite query performance, they suffer when updates to the views are required. In this respect, *view adaptation* is a crucial component in these systems. Furthermore, this view adaptation process strongly benefits from a fragmented approach to materialisation. To date, no research effort has attempted to combine the fragmented approach to XML data.

Finally, we provide details of the structure of this dissertation. We continue in Chapter 2 by giving a detailed literature review of existing work in this area; in Chapter 3, a system

overview is provided together with a description of the XML dataset used for examples through this dissertation; Chapter 4 presents our XFM view model; Chapter 5 introduces the containment checking algorithm; we then present our fragment selection mechanism in Chapter 6, which analyses existing views and determines those parts of the views to be materialised; we describe view changes in Chapter 7 together with the corresponding view adaptation algorithms; in Chapter 8, we present our evaluation with a detailed analysis of experiments; and finally, we provide conclusions and discuss future work in Chapter 9.

Chapter 2

Literature Review

The primary focus of our research is on view adaptation for materialised data. In this chapter, we begin with an overview of the adaptation process. We then examine adaptation in relational database systems, beginning with the earliest efforts, and discuss how issues have evolved over time. We complete our discussion in the relational context with an analysis of a multi-view based adaptation approach. We then proceed to discuss XML based adaptation. At the end of this chapter, we discuss two essential problems that are related to the multi-view based adaptation approach: containment checking and fragment selection.

2.1 View Adaptation Overview

View adaptation has been studied extensively in relational database systems for both centralised and distributed environments. The traditional *view maintenance* approaches, both in relational context [GMS93, LSK07, ZLE07] and in XML context [OCMH05, STP+05], aim to maintain the materialised view in response to the modifications of source data, whereas, the view adaptation approach aims to "adapt" the view incrementally in response to changes made in the view definition. When incremental adaptation is not possible, rematerialisation is required. As will be shown later, to reduce the cost of rematerialisation, different approaches are presented to reuse existing materialised data so that access to the source data can be avoided.

On the other hand, view adaptation is very close to the *query answering* [TYÖ+08b, WTW09] problem which aims to find a rewriting plan over an existing materialised view so that the

Customer($\underline{CustomerID}$, $\underline{AddressID}$, FirstName, LastName)
Address($\underline{AddressID}$, Street, City, State, ZIP)
Orders($\underline{CustomerID}$, $\underline{OrderID}$, $\underline{AddressID}$, OrderDate, OrderTotal)
OrderLine($\underline{OrderLineID}$, $\underline{OrderID}$, $\underline{ItemID}$, Status, Discount, QTY)
Item($\underline{ItemID}$, Title, Cost, Description)

Figure 2.1: TPC-W Benchmark

existing materialised data can be reused to answer the new query. However, they differ from each other based on their purpose, the cost of the process and the approach adopted as shown in Table 2.1.

In the rest of this chapter, we introduce view adaptation approaches within different contexts. We analysis different approaches in detail with examples based on the database schema depicted in Figure 2.1, where attributes that are underlined indicate the key attributes. This sample schema is a subset of the schema introduced in the TPC-W Benchmark [TW05] which is a transactional web benchmark.

2.2 Early Efforts at View Adaptation

In [Bel98], view adaptation is triggered by view redefinitions that are caused by *schema evolution*. In their approach, views are defined based on an integrated schema as shown in Figure 2.2. The integrated schema is generated by merging source schemas obtained from different data sources. View redefinition is implicitly caused by the schema evolution of the existing views as explained below.

Suppose the clients or the data warehouse administrator can execute update upon the data warehouse schema independently of the information sources (e.g., *Data Source 1*). As a result, the modification and deletion of an attribute type in an existing view schema will cause the incoherence with regard to the data source schema.

Their solution to the modification of an attribute type in a view schema is to create a new view with the new attribute type without materialising it, but reference to the old materialisation. When deleting an attribute, they apply a `Hide` operation to hide the old materialised data corresponding to the deleted attribute. In both case, no new materialised data is produced.

	Query Answering	View Adaptation
Problem	Given a materialised view V and a query Q, how Q is answered by using V.	Given a materialised view V and a change c, V' is the view after applying the change c to V. The problem is how V is updated by reusing the previously materialised data.
Approach	1. To find a query C, upon Q and V, when executed on V, gives the result of Q. The query C is referred to as the *compensation query* or the *composing query* 2. *Recomputing*: if C cannot be found, then compute Q from scratch.	1. *Incremental Adapting*: If V' is less restrictive than V, then the process adds *extra* data into the materialised view. If V' is more restricted than V, then the process removes *redundant* data from the materialised view. 2. *Recomputing*: if previously materialised data cannot be reused, then compute V' from scratch.
Cost Complexity	The complexity of using V is $O(n)$, where n indicates the amount of data contained in V.	The cost of the maintenance process is $O(\log(n))$ by the *incremental adapting*.
After Execution	V is retained.	V is deleted and V' is kept.

Table 2.1: Query Answering and View Adaptation

Summary and Issues. In their approach, a limited amount of changes are supported, modifying or deleting an attribute. View adaptation is achieved by either referencing to or *hiding* the old materialised data, the old materialised data is not actual adapted.

The work presented in the next section takes different types of changes into account. Different view adaptation algorithms are provided to handle each type of change and the adaptation process is treated as an update process over the existing materialised view.

2.3 View Redefinition in SQL Clauses

Different types of changes are considered by [GMR95, GMRR01] for relational database systems in a centralised environment. In their approach, additional information is stored with materialised views to facilitate view adaptation. Views are defined by the SELECT-FROM-WHERE queries and changes can be made to either the SELECT, FROM or WHERE clauses. Examples of possible changes are: adding or deleting an attribute in a SELECT

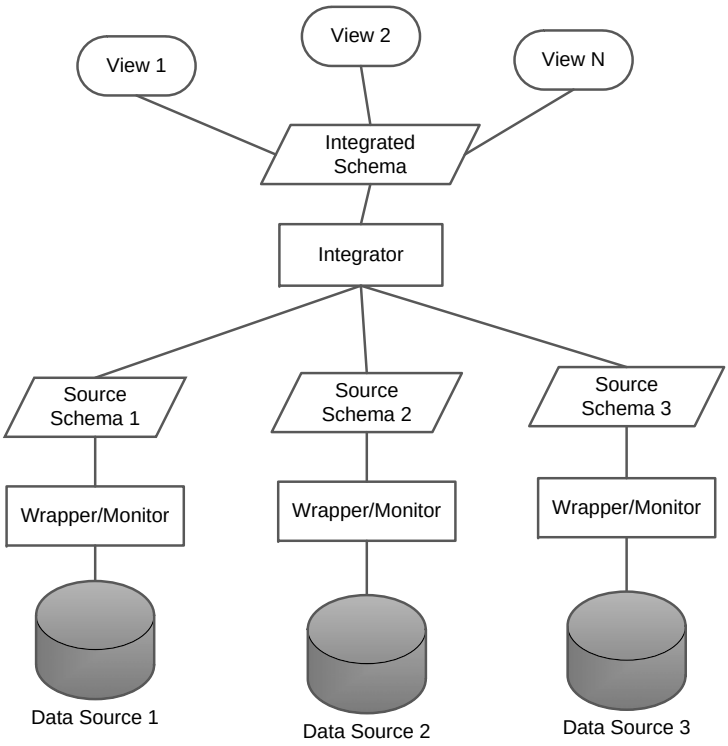


Figure 2.2: A Schema Architecture of a Data Warehouse

clause; adding, deleting, or modifying a predicate in the WHERE clause; adding or deleting a join operand in the FROM clause. Our examination of this work is useful in presenting a more in-depth overview of the issues involved and how they can be addressed.

When changes are encountered, the view adaptation process is treated as an incremental update problem upon the old materialised view, which means that the view adaptation is achieved by executing an update statement upon the old materialised data. The update statement is obtained by comparing the new and old view definitions and the result of the new materialisation is computed by executing the update statement over the materialised data. However, in most of the cases, the update statement needs access to the source data as the old materialised data do not contain sufficient information, e.g., if the primary/foreign key is not stored, the process cannot identify rows in a view (table). As a result, extra attributes are stored with the view to facilitate the adaptation process. Those extra attributes could be the entire set of attributes of a relation or merely the primary/foreign keys. As demonstrated in the examples listed below, when the SELECT clause is changed, in order to achieve the adaptation process, it is necessary to store the key attributes with the view.

Example 2.1 (The Old View V)
CREATE *VIEW V AS*
SELECT *Title, OrderDate*
FROM *Item* & *OrderLine* & *Orders*
WHERE *Cost>100* ■

Example 2.2 (The New View V)
CREATE *VIEW V' AS*
SELECT *Title, OrderDate, Description*
FROM *Item* & *OrderLine* & *Orders*
WHERE *Cost>100* ■

Example 2.3 (Changing the SELECT Clause - The Update Statement)
ALTER TABLE *A* ***ADD*** Description ***UPDATE*** *A* ***SET*** Description =
*(****SELECT*** *Description* ***FROM*** *Item* ***WHERE*** *Item.ItemID=G.ItemID)* ■

Define a view V which stores title and order date of all items that cost greater than 100 Euro. The view is expressed by the SQL statement shown in Example 2.1, where the "&" is used here to indicate the equality conditions in a natural join (used throughout this chapter). Suppose one would like to add a new attribute, *Description*, into the SELECT clause, V then becomes V' as shown in Example 2.2. In this case, since the foreign key is not stored with V, e.g., *ItemID*, the view adaptation process cannot identify each *item* stored in V. Therefore, the approach is to augment V with attribute *ItemID*. Suppose the augmented view is stored in a relation *A*, then the adaptation problem can be treated as an incremental update problem by executing an update statement shown in Example 2.3 upon the relation *A*.

Example 2.4 (Changing the FROM Clause - The Old View V)
CREATE *VIEW V AS*
SELECT *FirstName, LastName, OrderTotal, AddressID*
FROM *Customer* & *Orders*
WHERE *Customer.CustomerID=Orders.CustomerID* ■

Example 2.5 (Changing the FROM Clause - The Update Statement)
ALTER TABLE *V* ***ADD*** *Street, City*
UPDATE *V* ***SET*** *Street, City = (****SELECT*** *Street, City*
FROM *V, Address* ***WHERE*** *V.AddressID=Address.AddressID)* ■

When a change takes place in the FROM clause, a relation is either added or deleted to/from the join expression. A two-step approach is presented for each case. For the case where a new relation is added into a view, the adaptation process first alters the view (table) by extending it with the extra attributes required and then executes an update statement upon the view to fill the columns corresponding to the new attributes. When a relation is removed, the adaptation process first deletes duplicate tuples from the old materialisation and then adds dangling tuples to the new view. A dangling tuple is a tuple in a base relation that may not join with any tuples in another base relation. Both cases require key attributes to be stored with the old view so that when a join relation is added (removed) into (from) the view, the adaptation process can identify extra data to be added into or, redundant data to be removed from, the old materialised data. Example 2.4 and Example 2.5 demonstrate the circumstance that when a join relation is added into the FROM clause.

Define a view V, as shown in Example 2.4, which contains customer names and total number of items that they have ordered. Suppose one would like to add the relation Address into the FROM clause with additional attributes *Street* and *City* into the SELECT clause, the adaptation is achieved by, 1) augmenting V with additional attributes required, *Street* and *City*; 2) filling the new columns with the corresponding data required for *Street* and *City*. Those steps are achieved by executing the update statement shown in Example 2.5 upon the old materialisation provided that key attributes are stored. In this case, the key *AddressID* is required to determine which rows containing the attributes *Street* and *City* should be retrieved so that they join with existing materialised data.

Summary and Issues. Several limitations appear in this approach:

1. This approach is limited to cases where foreign keys are available in the database schema. However, in a real world environment this might not be the case.

2. The incremental update approach requires foreign keys (or other attributes) always stored with the view, which is fine for small number of attributes. When there are several base relations and many attributes exist, keeping a copy of all may not be feasible. In the worst case, an entire table is stored.

3. The adaptation algorithms do not take network communication cost into account.

CustomerID	*AddressID*	*FirstName*	*LastName*
c1	a1	f1	l1
c2	a2	f2	l2
c3	a3	f3	l3
c4	a4	f4	l4
c5	a5	f5	l5

(a) Customer

CustomerID	*OrderID*	*AddressID*	*OrderDate*	*OrderTotal*
c1	o1	a1	25/03/2010	1
c1	o2	a1	04/02/2010	2
c3	o3	a3	01/01/2010	5
c2	o4	a2	10/01/2010	2
c2	o5	a2	15/01/2010	1

(b) Orders

Table 2.2: TPC-W Benchmark: Sample Data

They assume that views and base relations are stored on the same site. However, in reality, it is likely to be the case that materialised views and base relations are stored separately over the network.

4. For the case of changing the FROM clause, their approach works only under a very restricted situation, when duplicates of tuples are maintained, dangling tuples are allowed and key attributes must be stored with the view.

2.4 View Adaptation Using Auxiliary Attributes

The work presented in [MD96] either extends or provides more efficient methods based on the centralised approach for data warehouses. They proposed a view adaptation mechanism that saves network communication cost, mainly the data transferring cost, and does not rely on foreign keys. Their approach augments base relations and derived materialised views with a *join-count* and a *derive-count* attribute, respectively. The join-count on a base relation indicates how many times a tuple joins with tuples in other relations and the derive-count attribute represents the number of derivations of each view tuple. They support the situation that dangling tuples are allowed.

Unlike the approach in [GMR95], which relies on the appearance of the foreign key attributes, in this approach, when changes take place in the SELECT clause, it stores **join attribute**s as extra information with views and with an extra join-count attribute indicating the number of times a tuple joins with other tuples in the second relation. Similar to the

CustomerID	*OrderDate*	*OrderTotal*
c1	25/03/2010	1
c1	04/02/2010	2
c2	10/01/2010	2
c2	15/01/2010	1
c3	01/01/2010	5

(a) Materialised View V

CustomerID	*AddressID*	*FirstName*	*LastName*	*join-count*
c1	a1	f1	l1	2
c2	a2	f2	l2	2
c3	a3	f3	l3	1
c4	a4	f4	l4	0
c5	a5	f5	l5	0

(b) Customer (Augmented with join-count attribute)

Table 2.3: Sample View and Augmented Relation

previous approach, the purpose of augmenting the view with extra data is to facilitate the identification of the relevant data required for adaptation and additionally, to reduce the network communication cost. Example 2.6 gives an illustration of a change takes place in the SELECT clause. Table 2.2 shows the sample data of the Customer and Orders relations that are used in Example 2.6.

Assume we wish to define a view V, which stores all details of the customer orders including customer id, order date and order total. The view is expressed in Example 2.6 and the materialised data of the view is shown in Table 2.3a.

Example 2.6 (Handle Changes with a Join-Count Attribute)
CREATE *VIEW V AS* ***SELECT*** *CustomerID, OrderDate,*
OrderTotal ***FROM*** *Customer* & *Orders* ■

Suppose one would like to add customer names into V. This involves retrieving all tuples from the Customer relation and joining them with the materialised view V. However, by observation, one would easily find that customer c4 and c5 (see Table 2.2) have no orders at all. Therefore, retrieving c4 and c5 from the base relation is a redundant process, which may become a significant cost when the number of redundant tuples are large and the *Customer* relation is stored on a different site on the network. By augmenting the base relation with an additional join-count attribute as shown in Table 2.3b, the process can avoid unnecessary data transferring cost caused by the dangling tuples, e.g., c4 and c5. The old view is then

FirstName	LastName	OrderTotal	derive-count
f1	l1	1	1
f1	l1	2	1
f2	l2	1	1
f2	l2	2	1
f3	l3	5	1

(a) Before Adaptation

FirstName	LastName	derive-count
f1	l1	2
f2	l2	2
f3	l3	1
f4	l4	1
f5	l5	1

(b) After Adaptation

Table 2.4: Change in FROM Clause

adapted by joining the old materialised view with tuples retrieved from the base relation and projecting out the new attributes that are desired, e.g., in this case, *FirstName* and *LastName*.

When removing a relation from the FROM clause, the previous approach [GMR95] works under very restricted parameters: when duplicate tuples are maintained and where dangling tuples are permitted in the view. As depicted in Example 2.7, the algorithm presented in this work does not have such restrictions as extra or redundant tuples can be identified by using the join-count and derive-count attributes.

Define a view V containing order information of all customers, the view is expressed in Example 2.7 where the materialised data contained in V is shown in Table 2.4a with an additional derive-count attribute.

Example 2.7 (Using Join-Count and Derive-Count Attributes)

CREATE *VIEW V AS* ***SELECT*** *FirstName, LastName, OrderTotal*

FROM *Customer & Orders* ■

Suppose one would like to delete relation Orders from V, which leads to the deletion of *OrderTotal* column in V as it is part of the relation Orders. The materialised view becomes less restricted after deleting and therefore, tuples that were previously eliminated should now be added into the view. In this case, customer c4 and c5 should be added. The following steps are used to achieve the adaptation:

- Execute the statement **SELECT** FirstName, LastName, derive-count **INTO** I **FROM** V. The purpose of this is to temporarily store attributes that will not be deleted into a relation I.
- Find tuples in Customer relation (Table 2.3b) where the join-count is **zero** and store them in a temporarily relation C, i.e., c4 and c5.
- Compute the new view $V' = I \cup C$; also update the derive-count (see cells in red in Table 2.4b);

The new materialisation results from the change is shown in Table 2.4b. As shown in Example 2.7 extra data transferring costs are eliminated when retrieving data from base relations and it is not necessary to retrieve data with join-count greater than 0 as they are already in the materialised view. The purpose of these examples is to demonstrate how this work addresses the shortcomings in [GMR95] where network communication costs were not considered. These costs are crucial in systems such as data warehouses as we will show in our experiments.

Summary and Issues. One of the problem involved in this approach is that, depending on the number of joins existing in a view, each join relation involved in the FROM clause is augmented by one or more join-count and derive-count attributes. When there are many relations involved in the query and each relation contains a large number of tuples, a significant amount of additional data is added into each relation. Additionally, views are treated as single entities and adaptation algorithms cannot detect and reuse exiting materialised data to further improve the adaptation performance or to reduce the number of accesses to the source data.

The next approach we are about to present considers the reusability of the old materialisation. Rather than augmenting the materialised view, their purpose is to find maximum reusability of the existing materialisation for the view adaptation process.

2.5 View Adaptation Using Expression Trees

The adaptation algorithms previously introduced [GMR95, MD96, GMRR01] focus on augmenting either materialised views, base relations or both, to facilitate the adaptation process. None studied the problem of how to maximise reuse of existing materialised data. The work proposed in [Moh97] focuses on identifying reusable materialised data. Only when changes are dramatic, extra data is then stored. In their approach, changes can be made at relational algebra level within an *expression tree*, e.g., adding/deleting/modifying relational algebraic operators. As their approach is based on the expression tree, we therefore refer to their adaptation approach as the *ExpressionTree* approach. Assume we require a view *CustomerOrder* containing customer order information after 10/02/2010. The view is represented by in Example 2.8.

Example 2.8 (A Sample View - CustomerOrder)

CREATE *VIEW CustomerOrder(FirstName,LastName,Item,Cost,Description) AS*

SELECT *FirstName, LastName, Item, Cost, Description*

FROM *Customer, Orders, OrderLine, Item*

WHERE *date>10/02/2010* ■

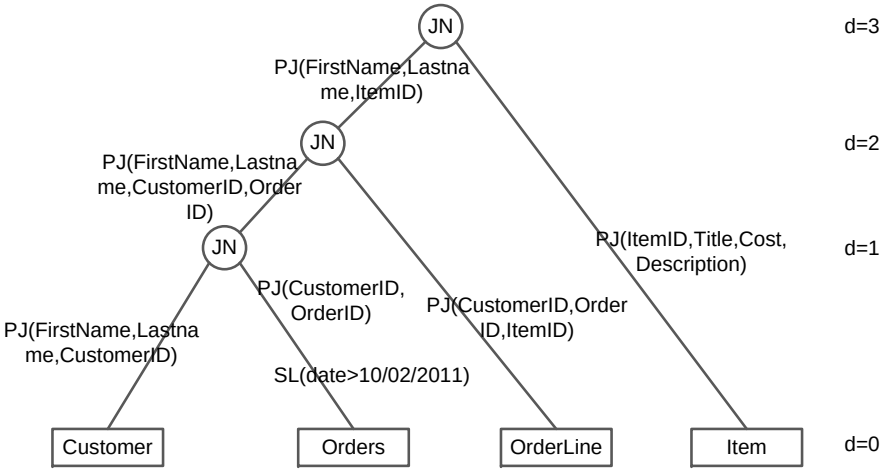


Figure 2.3: The Expression Tree of the View CustomerOrder

The *ExpressionTree* approach defines views using relational algebraic expressions containing Selection (`SL` or σ), Projection (`PJ` or π), Join (`JN` or $\bowtie$), Union (`UN` or $\cup$) and Difference (`DIFF` or $-$). The evaluation of a view can be represented by a binary expression

tree, where leaf nodes represent base relations and non-leaf nodes contain binary algebraic operators. The unary operations such as selection and projection are associated with edges of the expression tree. Figure 2.3 gives an example of the expression tree corresponding to the view, CustomerOrder, in Example 2.8. The depth of each node within the expression tree is denoted by d. The depth of leaf nodes is 0, whereas, the depth of a non-leaf node is defined as the maximum depth of its descendants + 1. Root node has the maximum depth value in an expression tree.

The result of each binary algebraic operator indicates the intermediate result of the view. For instance, in Figure 2.3, the node at d=1 represents the intermediate result of performing a selection (*SL(data>10/02/2010)*) over the base relation Orders followed by a join operation between Customer and the derived tuples of Orders generated by the select operation. The result produced by each operator serves as the input to the next operator located above it. The result of the view is obtained by computing the root node of the expression tree. When changes are made to the operators, it is not necessary to recompute all the operators from scratch.

From \ To	DIFF ($-$)	UN ($\cup$)	JN ($\bowtie$)
DIFF ($-$)	n/a	$V' = V \cup IR(\downarrow c_2)$	$V' = IR(\downarrow c_1) \bowtie IR(\downarrow c_2)$
UN ($\cup$)	$V' = V - IR(\downarrow c_2)$	n/a	$V' = IR(\downarrow c_1) \bowtie IR(\downarrow c_2)$
JN ($\bowtie$)	$V' = IR(\downarrow c_1) - IR(\downarrow c_2)$	$V' = IR(\downarrow c_1) \cup IR(\downarrow c_2)$	n/a

$\downarrow c_1$ and $\downarrow c_2$ represent all left and right parts of nodes of the root node, respectively.

Table 2.5: Changing the Root Node

If one changes the operator of the root node to the join operator or from a join operator to another operator, as shown in Table 2.5, the old materialisation cannot be used for adaptation and intermediate results must be stored with the view. IR returns the intermediate results of an algebraic expression. When the operator changes from union to difference or vice-versa, then the process can reuse the left part of the root node and recompute only the right part.

Summary and Issues. As with previous approaches, the *ExpressionTree* approach treats views as single entities with no sharing considered. As views are defined using algebraic expressions encapsulated in nodes, it will be more beneficial if nodes can be somehow shared between views to further improve the adaptation process.

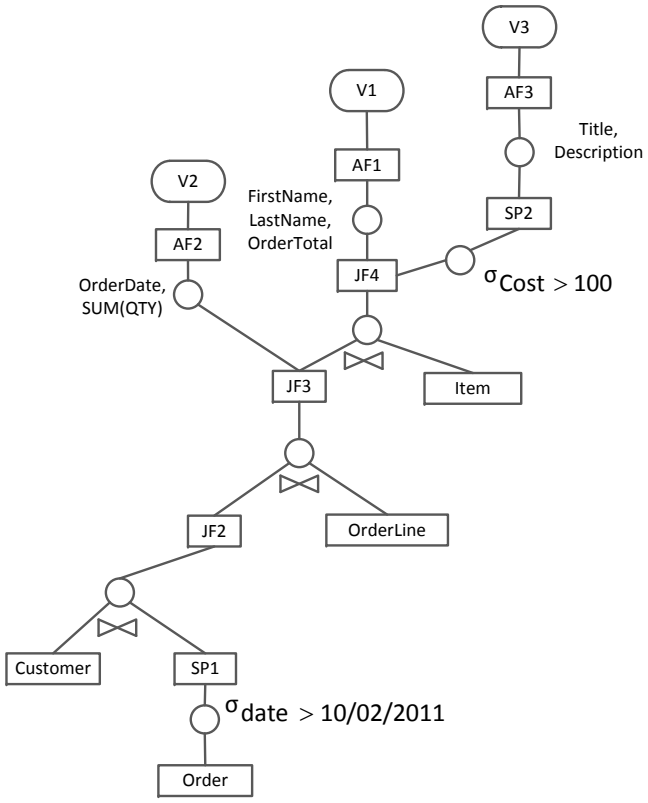


Figure 2.4: Multi-View Materialisation Graph

2.6 Fragment-Based View Adaptation

The *MultiView* approach proposed by [Bel00, Bel04] considers adaptation in a multi-view based environment, which provides the ability to reuse materialised data. Views are expressed by algebraic operators and fragments and common parts between views are shared and then exploited by the view adaptation process to improve the performance.

In their approach views are modelled using a *Multiview Materialisation (MVM) Graph*, which is a bipartite directed acyclic graph with two types of nodes: AND-nodes and OR-nodes. AND-nodes represent algebraic expression corresponding to select, project and join with possible aggregate function and OR-nodes represent the results produced by evaluating the algebraic expressions. As shown in Figure 2.4, the OR-nodes are represented by circles and AND-nodes are expressed by the rectangular boxes. An OR-node represents either a fragment or a base relation. The fragment is a node resulting from the application of an algebraic operator. There are three types of fragments, **SP_fragment**, **J_fragment**

and **A_fragment**, correspond to select, join and project operation, respectively. The leaf nodes of the MVM graph represent database relations, see Figure 2.4. Each fragment is a potential candidate for materialisation. Any change applied to an existing materialised view is reflected by changing the MVM graph.

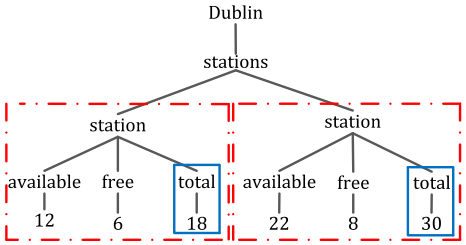
When changes are encountered, the *MultiView* approach adapts both the structure of the MVM graph and the materialised data affected by the change. Based on the actual type of fragment that is effected by the change, a corresponding view adaptation algorithm is applied. The main advantage of the *MultiView* approach is that when view definitions change, it is generally not necessary to update the entire view. Instead, only those fragments affected by the view change, representing a far smaller segment of materialised data, must be updated.

Summary and Issues. The *MultiView* approach was designed for relational database systems and does not suit other data models. When it comes to XML, different challenges are encountered, e.g., XML data is nested, irregular, heterogeneous and ordered, whereas, relational data is flat, regular, homogeneous and unordered. Additionally, as shown in §2.8 and §2.9, the two subproblems of the multi-view based adaptation mechanism, *containment checking* and *fragment selection*, are also facing different challenges for XML databases.

2.7 XML View Adaptation

In [AML+07], the authors proposed the first view adaptation mechanism for XML. However, rather than adopting the view adaptation technique to XML data warehouses, they apply the technique to the context of XML-based security. In their approach, an XML document is represented as a hierarchy of nested nodes with fine-grained access control applied to it at the node level. Access to XML data (nodes) is granted/restricted by defining a set of *positive/negative access control rules*, denoted by ACR^+ and ACR^-, respectively. The access control rules are represented by a subset of XPath expressions containing only ancestor-descendant and parent-child relationships. Views (also called *access control view* in their work) are defined by a combination of positive and negative access control rules. Figure 2.5b is an example of the access control view containing one positive rule and one

negative rule. The positive rule grants node access in an XML tree to users (see dashed box in red in Figure 2.5a). The negative rule prevents XML nodes from being accessed by the users (see solid box in blue in Figure 2.5a). The ***difference*** between ACR^+ and ACR^- are the result of the view.



(a) XML Data

$\text{ACR}^+ = \{\text{//Dublin// station}\}$

$\text{ACR}^- = \{\text{//Dublin//station/total}\}$

(b) Access Controlled View V

Figure 2.5: Access Controlled XML Data and View

Four types of view definition changes are allowed: (1) removal of a positive rule; (2) addition of a positive rule; (3) removal of a negative rule; and (4) addition of a negative rule. Depending on adding/deleting a positive/negative rule, nodes are either added to, or delete from, the old materialised data. Their view adaptation algorithms are based on a set of adaptation rules that are discussed later in this section. Furthermore, in their approach, adding or deleting data is achieved by using *deep intersect* [LLLL04] and *deep except* [LLLL05] operators, denoted by $\cap^D$ and $-^D$, respectively. The result of the *deep intersection* operation is the intersection between two subtrees of an XML tree, whereas, the *deep except* operation computes the difference between two subtrees.

Addition of Positive Rules. Two rules are used in response to the addition of a positive rule R:

- Containment Rule. If R is contained in $\text{ACR}^+ \cup \text{ACR}^-$, then no adaptation is required as R is covered by existing rules;
- Default Rule. If the containment rule does not apply, then R potentially adds extra data into the old materialised view. The process computes the extra data that are granted access by R and augments the previously materialised data. The default rule

is expressed to the following equation:

$$V'(\mathbb{D}) \quad = \quad V(\mathbb{D}) \cup (R(\mathbb{D}) -^{D} (R'(\mathbb{D}) \cup R''(\mathbb{D})))$$

where R' = $R \cap^{D}$ ACR^{+} and R'' = $R \cap^{D}$ ACR^{-}, $\mathbb{D}$ is an XML document.

Removal of Positive Rules. Deleting a positive rule R from ACR^{+} is handled by two rules:

- Containment Rule. If R is contained in ACR$^{+}{}_{new} \cup$ ACR^{-}, where ACR$^{+}{}_{new}$ is the new positive rule set after deleting R, then no adaptation process is required as deleting R from ACR^{+} does not remove any data from the materialised view.

- Default Rule. If R is not covered by existing access control rules, then the default rule is applied to remove data that were previously granted access by R. The process first determines what data were previously granted access and should be removed now and it then removes the data from the old materialised view.

$$V'(\mathbb{D}) \quad = \quad V(\mathbb{D}) -^{D} (R(\mathbb{D}) -^{D} (R'(\mathbb{D}) \cup R''(\mathbb{D})))$$

where R' = $R \cap^{D}$ ACR$^{+}{}_{new}$ and R'' = $R \cap^{D}$ ACR^{-}.

Addition of Negative Rules. The addition of a negative rule causes old materialised data to be either more restricted or may possibly have no effect. Three adaptation rules are used in response to the change caused by adding a negative rule: Intersection Rule, Containment Rule and Default Rule.

- Intersection Rule. If a new negative rule R does not intersect with any positive rule, $R \cap^{D}$ ACR^{+} = $\emptyset$, then adding R does not restrict access to the old view and, therefore, no adaptation is required.

- Containment Rule. If R is contained by a negative rule in ACR^{-}, then removing R does not prevent users from accessing the materialised view and, thus, no change is required.

- Default Rule. The same Default Rule as defined in the case of **Removal of Positive Rules** is applied.

Removal of Negative Rules. Removing a negative rule R is also handled by three adaptation rules: Intersection Rule, Containment Rule and Default Rule.

- Intersection Rule: if R does not intersect with existing positive rules, $R \cap^D$ ACR$^+$ = $\emptyset$, then removing R does not restrict access to the materialised data and, therefore, no adaptation is required.

- Containment Rule. If R is covered by ACR$^-{}_{new}$, where ACR$^-{}_{new}$ is the set of remaining negative rules after deleting R, then the removal of R does not effect the materialised view as there is another negative rule in ACR$^-{}_{new}$ that restricts access to the same data as R.

- Default Rule. If R is not covered by another negative rule in ACR$^-{}_{new}$, then data that were previously restricted by R are added. The adaptation process computes those extra data by intersecting R with all existing positive and negative rules. The new materialised data is the union of the previously materialised data and those extra data computed. The default rule is expressed by the following equation:

$$V'(\mathbb{D}) \quad = \quad V(\mathbb{D}) \cup (R'(\mathbb{D}) -^D R''(\mathbb{D}))$$

where R' = $R \cap^D$ ACR$^+$ and R'' = $R \cap^D$ ACR$^-{}_{new}$.

Auxiliary Rule Views. In order to improve the performance of view adaptation, a set of *auxiliary rule views* are created for each negative rule to facilitate the adaptation process. For the case that a negative rule is removed, data may need to be added to the view and typically it is necessary to find out the additional data that needs to be added to the view. The auxiliary rule view corresponding to each negative rule is used to identify those additional data.

Summary and Issues. There are several issues/limitations involved in this approach as listed below:

- Only a very limited amount of changes are supported and it is difficult to incorporate new changes. Only adding or deleting an access control rule (XPath expression) is supported.
- The cost of computing extra or redundant data is expensive, which requires the process to perform join operations between the change and all existing positive and negative rules.
- To create auxiliary rule views, one needs to materialise all negative access control rules of the view, which is impractical and leads to materialising a significantly number of data and, hence, many duplicate data are created.
- Finally, their approach does not support multi-view based adaptation where materialisation is shared between views.

2.8 Approaches to Containment Checking

Two essential problems that are involved in a multi-view based adaptation approach (as presented in [Bel04]) are: how to identify common sub-expressions between views and the extent of the change between the new and old views; and how to determine those view segments that must be materialised. The first issue is closely related to the *query containment* problem, whereas, the latter one is considered as a selection problem over existing views. Although those problems have been well researched in the relational context, e.g., [FTU98, FTU99] for the containment problem and [CHS02, Bel04] for the selection problem, different challenges are encountered for XML. In the remainder of this section, we will examine related research in containment checking for XML views, and in the following section, we will examine fragment selection.

Existing efforts for the containment problem are based on a subset of XPath expressions, mainly on two most important axes, child and descendant axes. Compared to the classical containment problem for relational conjunctive queries, the challenge for containment in XPath is that queries might involve recursion (e.g., queries may require navigation along the descendant-axis). The first attempt at containment checking for XPath queries was proposed by [MS02, MS04]. They proposed two techniques, *canonical model* and *homomorphism*,

both covered in this section.

The general concept of containment between two XPath expressions is that the evaluation result of the first expression over an XML tree is contained in the result of the second expression. Thus, the first expression is said to be contained by the second one. To verify the containment relationship, it is necessary to determine that for all trees, the evaluation result of the first expression is always contained by the second one. It has been shown in [MS02] that it is sufficient to find a *counter example* where the evaluation result of the first expression over an XML tree is not contained by the evaluation result of the second one. However, as there may be an infinite set of trees [MS04], it is thus, necessary to reduce the search space. The *canonical model* approach reduces the search space of the containment checking from an infinite set of trees to a finite set. However, the search space is still very large which leads to an exponential-time algorithm for checking containment.

The *homomorphism* approach provides a much more efficient mechanism for containment checking. However, it is an incomplete algorithm, which means that the existence of the homomorphism is not a necessary criterion for containment, as it may return *false negatives*. Besides the canonical model and homomorphism, there is also *automata* based technique [Nev02, NS03], which is based on *tree automata*. The idea of the automata approach is to find a set of *all* counter examples, where a containment relationship does not exist. If this is not a NULL set, it is represented by a tree automaton. At the start of containment checking, the process constructs an automaton for the first expression representing all trees from which the result of the expression is obtained and it then builds an automaton for the second expression representing all trees that no result is returned when evaluating the second expression over them. The containment relationship is verified by joining the two automata and checking whether the join returns an empty set. The process returns an empty set if containment relationship exists, otherwise, a non-empty set is returned. While this approach provides a complete containment checking algorithm, it requires exponential time for processing that is not feasible for practical applications.

As the consequence, the mainstream of the existing research tries to narrow the gap between canonical model and homomorphism approaches, that is to provide an approach which is more efficient than the canonical model approach and more complete than the homomorphism approach.

To extend the homomorphism based approach, existing efforts focus on the containment problem in the presence of the DTD or XML Schema file, e.g., the *chase* technique [Woo01, AYCLS02, Woo03], where the containment relationship is checked against the constraints outlined by the DTD or XML Schema. However, although those approaches derive the advantage of the homomorphism approach, they also suffer from the disadvantage brought by the homomorphism. Another problem of those approaches is that recursion may be defined in a DTD or XML Schema file and exists in an XML tree, however, the depth of the recursion is unknown which makes containment difficult to be verified.

The *conditioned* and *hidden conditioned* approaches proposed by [FLZ07] also extend the homomorphism technique, the algorithms they provided are complete under the conjunction of some conditions. The problem of the conditioned homomorphism is that there are special cases that still return false negatives. The hidden conditioned homomorphism covers those special cases and provides a more complete algorithm. However, both approaches need to compute all potential conditions that might be required to satisfy the containment relationship. Due to the lack to metadata information, e.g., constraints covered by DTD and XML Schema files, some of the computed conditions are redundant.

The *summary-based* [ABMP07] approach provides containment checking under constraints outlined by a strong DataGuide [GW97] of tree-structured data. The benefit of a summary based approach is that it provides more precision regarding the structure of the XML data and explores the exact depth of the recursion defined in the DTD or XML Schema files. However, the containment algorithm they provide is concerned only with the *path constraint*s. The path constraint restricts the node to be processed must satisfy the root-to-node path defined in the constraint. A similar approach is also presented in [LWH10] which determines the equivalence between two tree patterns in the presence of a DataGuide. However, neither approach takes *subtree constraint*s into consideration. The subtree constraint restricts nodes to be processed need to have the exact subtree structure defined in the constraints. As a result, both approaches may lead to incorrect result of the containment checking.

Summary and Issues. Containment checking with constraints, reduces the search space by avoiding unnecessary checking. However, the approaches discussed here, are based

on XML schemas, DTD or a path-based data guide. Here, only root-to-node paths are considered and the potential subtree constraint is ignored. Both the search space required by the containment checking algorithm, and the performance should be further optimised by taking a more comprehensive set of constraints into account. For example, the subtree structure and ordering of nodes can significantly reduce the search space for containment checking.

2.9 Approaches to Fragment Selection

Apart from the containment problem, a multi-view based approach also suffers from the issue of determining what part of the views should be materialised to avoid duplication and to expedite the view adaptation process. Existing selection mechanisms fall into three parts which we refer to them as *plan-based selection*, *full materialisation selection* and *partial materialisation selection*, respectively.

Many research efforts have been proposed for cost-based plan selection in relational database systems. In [GPSH02], the authors provide a cluster based selection plan, where queries are grouped based on their corresponding similarities. In their approach, they store a database of plans and attempt to assign one of these plans to the new query with the expectation that the selected plan would be the same as the plan generated by the optimiser. Only in the event that no suitable assignment can be found, is the optimisation process actually carried out and the newly generated plan is added to the *plan database* for future use. In [GCV09], the authors present a cost-based optimisation and execution framework. Their selection algorithm is based on a navigation-focused XPath algebra with novel operators and a comprehensive set of rewriting rules. By evaluating the costs of different algebraic representations after applying rewriting rules, their approach can select best plans for query execution. However, while these approaches provide an example of selecting the appropriate view to answer a query, they do not solve the problem of selecting views for *materialisation*.

On the other hand, there has been considerable research on full materialisation selection for relational database systems and this can be broadly categorised in two ways: centralised approaches [GZ08, KGJ10] and distributed approaches [BL03, YGYL05, CBHB09]. The former chooses materialised views in centralised scenarios, where storage is considered

to be the limiting factor while the later chooses materialised views in distributed environments, where the primary factor for concern is network communication costs. However, all approaches focuses on selecting entire views for materialisation and do not consider the potential shareability between views.

The partial materialisation mechanism discussed earlier in this chapter [Bel04], tries to balance the query processing cost and view maintenance cost. In their approach, each OR-Node (fragment) in the view graph is associated to a level, where the top OR-node has the level value 1. When an OR-node at level 1 is materialised, the query processing cost is low as the result of that OR-node is the result of the view, but the maintenance cost is high as the entire result is stored. However, if the OR-node at any other level is materialised, then the query processing cost is relatively high, but the maintenance cost is low. The purpose of their approach is to find an intermediary level for materialisation which balances the cost of query processing cost and view maintenance. The selection algorithm is based on two measures: *local benefit* and *global benefit*. A view is considered locally beneficial if its materialisation significantly reduces the query processing cost without significantly increasing the view maintenance. The global benefit is the measure of the importance of a fragment to all views in the view graph. Their selection is a two-step process, where the first step is to select a set of fragments within each query based on the *local benefit* and the second part filters fragments selected in the first part according to their global benefit to the entire view graph. In their approach, a suboptimal decision may be made when materialised views are dissimilar to each other, leading to potentially poor performance during view adaptation.

Summary and Issues. When there are a large number of views, merely calculating the benefit of each fragment to all views is not an appropriate solution. A more desirable approach would take user preferences into account. Suppose the benefit of a fragment is high, but it is only shared by views that are rarely required by the users, therefore, there is no reason to materialise such a fragment. Besides, views should only compare to other existing views that are similar to them as this provides a more accurate measure of the benefit that a fragment contributing to existing views. As we will shown in our experiments, view adaptation performs better by attempting to optimise the selection fragments for materialisation.

2.10 Summary

In this chapter, different view adaptation approaches were examined and analysed. Early work failed to "adapt" materialised data, and instead simply hid the old materialisation from the new view, or referenced the old materialisation when attribute type is changed. The centralised approach discussed in §2.3 provided a more comprehensive view adaptation mechanism that could be used to handle different changes. However, they did not take network communication costs into account, a key feature in §2.4. This work utilised two auxiliary attributes to save the cost of transferring redundant data over the network. To maximise reuse, the *ExpressionTree* approach divided views into different nodes and analysed when old materialisation can be reused and when intermediate results are required.

The *MultiView* approach was shown to outperform the single view based approaches by sharing materialised data between views to improve not only the view adaptation performance, but also the reusability of materialised data. Additionally, the cost of maintaining materialised data is also reduced. However, existing XML view adaptation does not employ a multi-view approach, with view adaptation algorithms based on single views, and changes are very restricted. Thus, views must often be recomputed with data retrieved from data sources. Strongly motivated by this literature review, the next chapter will provide an outline of a fragment based adaptation system for XML repositories.

Chapter 3

The XFM View Framework

In Chapter 2, a number of view adaptation approaches were discussed in both relational and XML contexts. Any approach to view adaptation requires a suitable framework and methodology in order to maintain consistency between views and underlying data. In this chapter, we present our *XML Fragment Materialisation* (XFM) view framework which provides an overview of the different processes developed during this research. Our motivation is that the reader understands the tasks involved in view adaptation. In §3.1, we introduce the framework in terms of workflow; in §3.2, we discuss the system architecture and provide more detail on the role of each process; and finally in §3.3, we present a detailed overview of the XML repository used in our evaluation as this will be used repeatedly, to explain concepts and features of our system.

3.1 View Adaptation Outline

The View Adaptation system is deployed on top of the XFM view framework which is built based on our view model introduced in Chapter 4, and it contains five major processes: *Graph Transformation*, *Classification*, *View Adaptation*, *View Selection* and *Materialisation*. As shown in Figure 3.1, these processes, depending on their objectives, are divided into two parts, the offline and online parts, as described below:

- The offline part follows arrows labelled with 1, it accepts XPath Queries as input (**Input 1**); constructs the overall view representation of those XPath queries (**P1**),(**P3**);

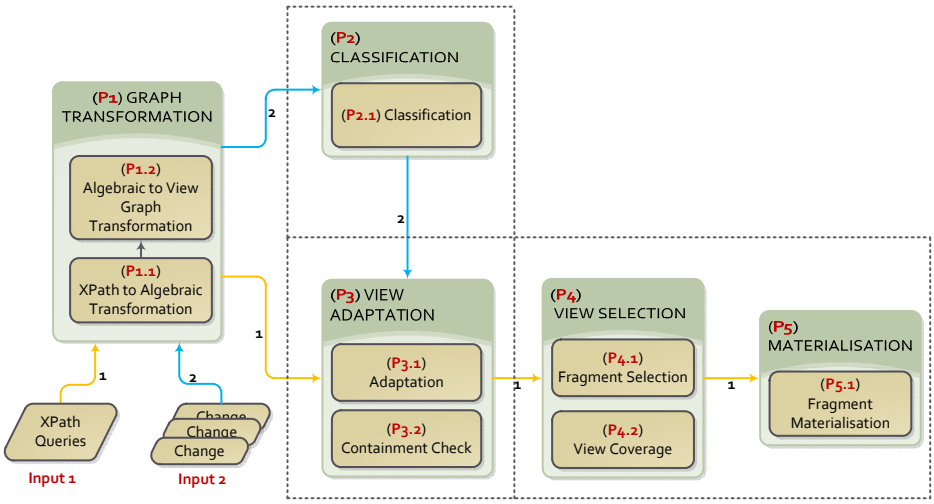


Figure 3.1: XFM View Adaptation: Process Flow

and selects candidate fragments for materialisation (**P4**),(**P5**).

- The online part follows arrows labelled with 2, which runs at all times and continuously accepts user requirements (changes) as input. Changes (**Input 2**) are transformed into the internal representation (**P1**), based on the type of changes applied (**P2**) and the process adapts views accordingly (**P3**).

In summary, the offline segment involves the initialisation of the View Adaptation system, which creates and integrates views, selects view fragments for materialisation and then materialises those selected candidates. Although, this runs only once, after a certain number of changes are made to views, the system administrator may decide to run this part again to reorganise the materialisation. The online part of the system, continuously accepts view change requests from users and is responsible for synchronising the materialised data and the new view definitions. In next section, we give a detailed description of the View Adaptation System architecture and its components.

3.2 The XFM View Adaptation System

As illustrated in Figure 3.2, the system accepts a set of queries as input and they are transformed into the *XML Fragment Materialisation (XFM) View Graph*, by the *Graph Transformation* (**P1**) process. In the XFM view graph, common parts are shared between different

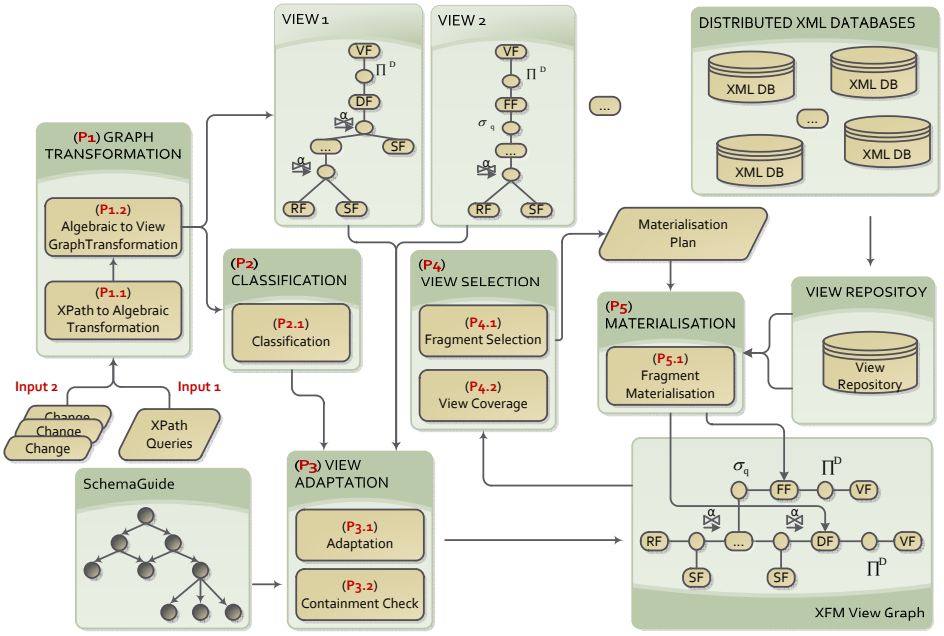


Figure 3.2: View Adaptation System Architecture

views by means of fragments. The *View Selection* (**P4**) process then selects fragments among the views for materialisation based on different costs estimated by the process and the *Materialisation* (**P5**) process materialises all fragments that are selected by the *View Selection* process. On the other hand, if changes are submitted by users, the *Classification* (**P2**) process verifies the type of the change applied, and the *View Adaptation* (**P3**) process verifies the extent of the changes between the old and new views, using the *Containment Check* process (**P3.2**) and eventually adapts existing views (**P3.1**) both structurally, in terms of the XFM view graph, and physically, where it is necessary to adapt the actual materialised data. We now proceed with a brief description of each process.

3.2.1 Graph Transformation

The *Graph Transformation* (**P1**) process is responsible for translating XPath queries into our view graph representation and it is divided into two subprocesses as listed below,

1. *XPath to Algebraic Transformation* (**P1.1**), which translates each XPath expression into its algebraic representation.

2. *Algebraic to View Graph Transformation* (**P1.2**), which transforms algebraic repre-

sentations to the view graph representations through a set of fragments indicating the results of each algebraic operation.

3.2.2 Classification

The *Classification* (P2) process verifies the type of the change required, as one of two types:

1. *Structural* change, which further includes
   (a) adding a step into a view (XPath expression)
   (b) removing a step from a view (XPath expression).
2. *Predicate* change, which contains
   (a) adding a predicate to a step
   (b) removing a predicate from a step
   (c) modifying a predicate of a step.

3.2.3 View Adaptation

The adaptation process is involved in both the online and the offline parts of the View Adaptation system as described below:

1. In the offline part, the View Adaptation (P3) process is used to build the XFM view graph. Basically, after the graph transformation process, a **global** XFM view graph is created by integrating all individual view graphs. The initial global view graph is built from scratch by integrating all fragments of a view sequentially into an empty view graph. The common parts between views are identified with the assistant of the *Containment Check* (P3.2) subprocess and only displayed once in the global view graph. We treat the construction of the graph as a special case of the view adaptation.
2. In the online part, when changes are required, depending on the type of the change, the *Adaptation* (P3.1) subprocess dynamically adapts the effected materialised views and repeatedly invokes the *Containment Check* (P3.2) subprocess to determine the extent of changes between the new view and the old view. The adaptation process

maintains the existing views both structurally, in terms of fragment changes and physically, where it is necessary to adapt the existing materialised data.

3.2.4 View Selection and Materialisation

The *View Selection* (**P4**) process selects fragments from the global XFM view graph for materialisation and it consists of two subprocesses:

1. *Fragment Selection* (**P4.1**), which performs a baseline scan over the XFM view graph to select an initial set of fragments from the XFM view graph for materialisation.

2. *View Coverage* (**P4.2**), which performs an iterative-based process to ensure that all views are covered by the selection.

The result of the view selection process is a *Materialisation Plan*, which contains candidate fragments for materialisation. The *Materialisation* (**P5**) process materialises those fragments selected.

Table 3.1 lists the chapters in which each process and its subprocessess are discussed.

PROCESS		SUBPROCESS		CHAPTER
(**P1**)	GRAPH TRANSFORMATION	(**P1.1**)	*XPath to Algebraic Transformation*	Chapter 4
		(**P1.2**)	*Algebraic to View Graph Transformation*	
(**P2**)	CLASSIFICATION	(**P2.1**)	*Classification*	Chapter 7
(**P3**)	VIEW ADAPTATION	(**P3.1**)	*Adaptation*	Chapter 4,7
		(**P3.1**)	*Containment Check*	Chapter 5
(**P4,P5**)	VIEW SELECTION	(**P4.1**)	*Fragment Selection*	Chapter 6
		(**P4.2**)	*View Coverage*	

Table 3.1: Coverage of System Processes

3.3 The Worldbikes Repository

In Chapter 1, we discussed the growing value of the XML data generated by web services and in particular, the sensor web. As our work focuses on large XML repositories generated by these types of services and applications, we choose a dataset, the WorldBikes repository, generated by an application developed in Dublin City University [MRS11] as part of the Smart City initiative. A bicycle sharing system is a scheme in which numbers of bicycles are made available for rental to inhabitants and visitors. Many cities have now deployed

a bike sharing scheme where people can rent and return a bicycle to and from various locations. Stations are equipped with sensors that monitor station status such as bicycle and stand availability. This information is published online to inform consumers about the status of each bike station.

In our system, sensor data, containing the current status of every station, across every city, is harvested every 60 seconds and stored in an XML database system. As shown in [MRS11], every 24 hours, the system harvests 1,440 sensor outputs per city, each representing the state of all stations concerning the number of bicycles or parking slots at the given time stamp. The motivation for this project was to have the ability to perform more complex analysis on the usage of these bicycles and the growing repository provides a useful platform for our research, as the dataset is very large and an OLAP style of analysis can require continuous query adaptation.

Cities	Country	No. of Stations	Data Collected (GB)
Dublin	Ireland	44	0.30
Brisbane	Australia	120	0.72
Luxembourg	Luxembourg	54	0.36
Bruxelles	France	180	1.15
Amiens		25	0.18
Besancon		30	0.21
Mulhouse		35	0.24
Nancy		25	0.18
Nantes		88	0.56
Rouen		20	0.14
Santander	Spain	14	0.11
11 Cities	5 Countries	635 Stations	4.15 GB

Table 3.2: Sources and Statistics of Sensor Data

As shown in Table 3.2, sensor data is obtained from 635 stations in 11 cities across 5 countries. The total amount of data that has been collected so far is 4.15GB and the size is still growing by roughly 200MB per week.

In Figure 3.3, we show the structure of the WorldBikes repository. The repository consists of 11 cities (e.g., Dublin, the character * indicates those cities that can occur multiple times in the repository) and each of them contains information as listed below:

- Each station (*station*) consists a number of parking stands for bicycles, the actual

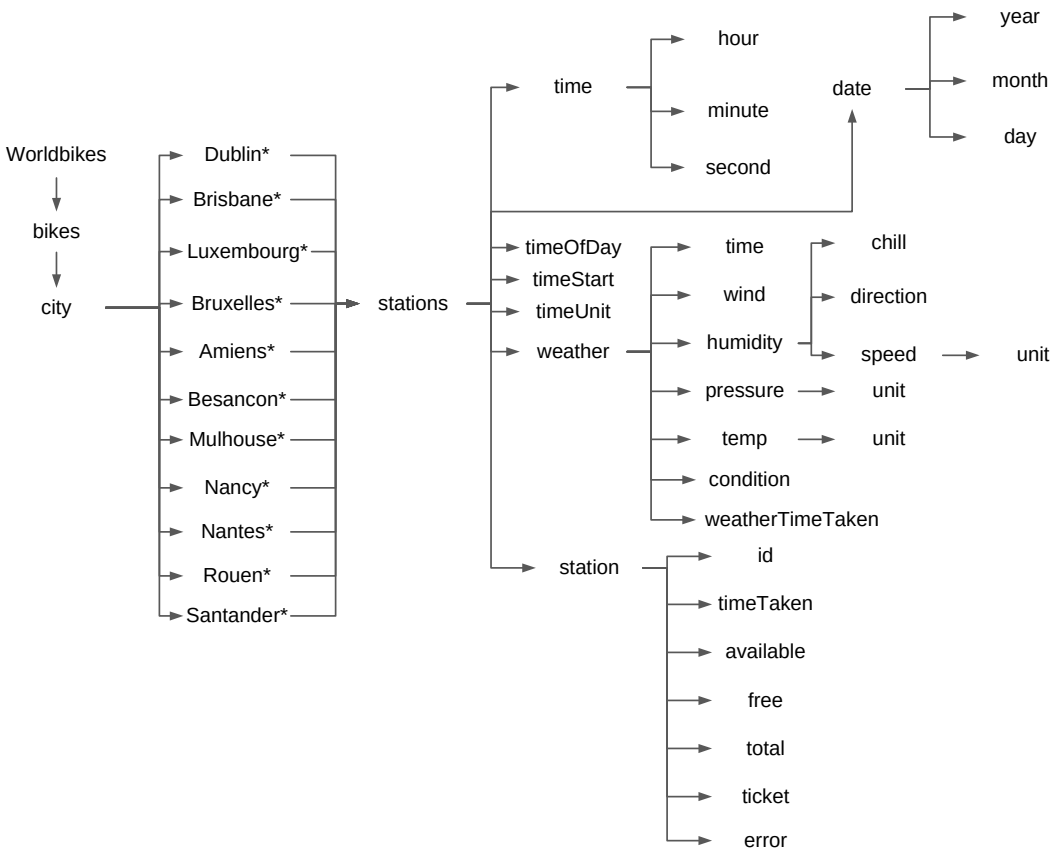


Figure 3.3: The Worldbikes Schema

bicycles available, and a sensor based system to determine the status of the parking stands (empty or occupied). The following information is made available online through the service provider's website,

- station ID (*id*)
- number of bikes available (*available*)
- number of free bike stands (*free*)
- total number of bikes (*total*)
- number of tickets (*ticket*)

The time required to retrieve the station data is also added (*timeTaken*) and any errors encountered during the collection are recorded (*error*).

- The station data is integrated with the most recent weather conditions (*weather*) including wind, humidity, pressure, temperature, condition with data obtained from

various weather sites.

- The time at which the information is retrieved is added as a timestamp to the station data, e.g., *date*, *time*, *timeOfDay*, *timeStart* and *timeUnit*.

```
<Worldbikes>
<bikes>
    <city>
        <Dublin>
            <stations>
                <time>
                    <hour>15</hour>
                    <minute>17</minute>
                    <second>25</second>
                </time>
                <date>
                    <year>2011</year>
                    <month>03</month>
                    <day>08</day>
                </date>
                <timeOfDay>15:17:25 08-03-2011</timeOfDay>
                <timeUnit>milliseconds</timeUnit>
                <timeStart>1299597445836</timeStart>
                <weather>
                    <time>Tue, 08 Mar 2011 3:01 pm GMT</time>
                    <wind>
                        <chill>41</chill>
                        <direction>230</direction>
                        <speed unit="mph">22</speed>
                    </wind>
                    <humidity>76</humidity>
                    <pressure unit="inches">29.91</pressure>
                    <temp unit="degrees farenheit">48</temp>
                    <condition>Mostly Cloudy</condition>
                    <weatherTimeTaken>70</weatherTimeTaken>
                </weather>
                <station>
                    <id>40</id>
                    <timeTaken>565</timeTaken>
                    <available>13</available>
                    <free>8</free>
                    <total>21</total>
                    <ticket>1</ticket>
                    <error>0</error>
                </station>
            </stations>
        </Dublin>
    </city>
</bikes>
</Worldbikes>
```

Figure 3.4: A Segment of the Worldbikes Dataset

A segment of the Worldbikes repository is shown in Figure 3.4, which lists the status of station 40 in Dublin with the corresponding weather conditions and timestamps. Below is a list of sample XPath queries:

- Q1. Return all information recorded for each station in the city of Dublin.

  //Worldbikes/bikes/city/Dublin/stations/station

- Q2. Return all stations in Dublin that have bicycle availability greater than 10.
  //Worldbikes/bikes/city/Dublin//station[./available$\geq$10]

- Q3. Return stations that had a wind speed greater than 22 mph.
  //stations[.//speed$>$22]/station

A larger query set is used in Chapter 8 as part of our evaluation process. In the evaluation, we use a query generator to define 1,000 views based on the Worldbikes Dataset. The next chapter introduces our data model and view graph which uses the same dataset in presenting a global view graph.

Chapter 4

The XFM View Model and Graph

The chapter begins in §4.1 by introducing the XML data model followed by a detailed description of the XML query language in §4.2. The XFM view model is then introduced in §4.3 together with a set of sequence-based algebraic operators in §4.3.1 and the corresponding translation process in §4.3.2, which translates XPath expressions into the algebraic representation. A set of fragments is presented in §4.4.1, together with the algebraic operators, form the XFM view graph as discussed in §4.4.2. We show how the algebraic representation is transformed into the view graph representation in §4.5. Finally, in §4.5.1, a construction algorithm is presented to demonstrate how a *global* XFM view graph is constructed based on a set of individual view graphs that are obtained by transformation process.

4.1 XML Data Model

An XML document can be modeled as a rooted and ordered tree with labels from an alphabet Σ. An XML tree corresponding to an XML document is denoted by t, where $t \in T_\Sigma$ and T_Σ denotes the set of all possible trees that are formed by labels obtained from Σ. An XML tree node corresponds to either an element, attribute or character content in an XML document. The root node of an XML tree represents the *document root* (root element) of the corresponding XML document. An XML tree is expressed as a 4-tuple, outlined below in Definition 4.1.

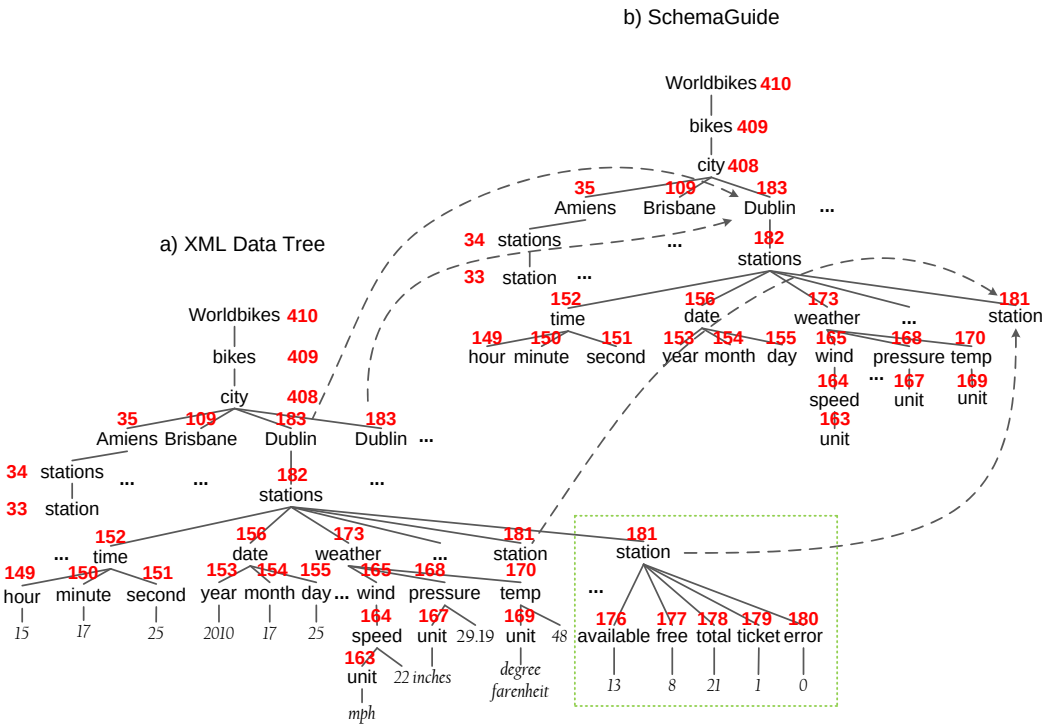


Figure 4.1: XML Data Tree and Schema Guide of Worldbikes Dataset

Definition 4.1 [XML Tree]

For a given XML tree t of an XML document, t is represented by a 4-tuple $<\mathbb{R}_t, \mathbb{N}_t, \mathbb{E}_t, \mathbb{L}_t>$, where $\mathbb{R}_t$ is the root of t; $\mathbb{N}_t$ denotes a set of nodes in t; $\mathbb{E}_t$ indicates a set of edges in t; and $\mathbb{L}_t$ represents a set of labels in t, where $\mathbb{L}_t \subseteq \Sigma$. ■

Suppose t is an XML tree corresponding to the Worldbikes dataset, Figure 4.1a demonstrates a segment of t. *Worldbikes* is the root of t represented by $\mathbb{R}_t$. By definition, $\mathbb{N}_t$ contains all nodes in t and $\mathbb{L}_t$ contains all labels of those nodes, e.g., the label of $\mathbb{R}_t$ in this case is *Worldbikes*. For a given node $u \in \mathbb{N}_t$, $\mathbb{L}_t(u)$ returns the label of node u. For example, if u is the root of the XML tree in Figure 4.1a, then $\mathbb{L}_t(u)$=*Worldbikes*, which is the label of the root. $\mathbb{E}_t$ contains a set of edges between nodes in the tree, for example, the edge between *Worldbikes* and *bikes*. In the rest of this dissertation, we refer to the node in an XML tree as an ***instance node*** and a node labelled with l as the node l, or simply l, e.g., a node labelled with *Worldbikes* is referred to as the node *Worldbikes*, or simply *Worldbikes*. We use them interchangeably.

Additionally, we denote $\mathbb{E}_t^+$ to be the transitive closure of $\mathbb{E}_t$ defining the ancestor-descendant relationship between any arbitrary pair of instance nodes within t that are not directly connected to each other, e.g., the edge between *Worldbikes* and *city*. Nevertheless, if two nodes are directly connected to each other in t, then their edge is specified in $\mathbb{E}_t$. A subtree of t is considered as the tree consisting of a node in t and **all** or **part** of its descendants. A subtree corresponding to a node u in t, is denoted by t_u. For example, the dotted box in Figure 4.1a is a subtree rooted at the node *station*.

For a path starting from the root node and ending at any non-root node, we refer to it as a *tree path* of that XML tree (the concept of the tree path is essential for proving the propositions used by our containment checking algorithm as discussed in Chapter 5). Each node within an XML tree has a tree path associated with it, which is the path starting from the root node of the XML tree and ending at the node itself. Two nodes are said to be on the same path if and only if the paths associated with them contain exactly the same node labels and they are in the same sequential order. A formal description of a tree path is outlined in Definition 4.2.

Definition 4.2 [Tree Path]

Given an XML tree t*, a* path P *is a chain of labels separated by / denoted by /*l_1*/*l_2*/.../*l_k*,* $k \leq n$*,* n *is the total number of labels, such that* l_1 *is the label of* $\mathbb{R}_t$ *and* l_2 *is the label of one of the child of* $\mathbb{R}_t$ *etc. A node* u *is on path* P *if the label path going from the root of* t *to node* u *is* P*. A node* u *is on the path of a node* v *if they are both on path* P*.* ■

Based on Definition 4.2, we can also say that a node u is said to be *reachable* from a node v by a path P, if and only if u is on the path P and the label of v corresponds to a label l in P, where the label l is not equal to the label of u ($l \neq \mathbb{L}_t(u)$).

Example 4.1 (Tree Path with Explanation)

Suppose a path P *is represented by the following expression,*

```
/Worldbikes/bikes/city/Dublin/stations/station
```

which indicates the path from the node Worldbikes *to the node* station *(see Figure 4.1a), where* bikes*,* city*,* Dublin *and* stations *are the intermediate nodes between* Worldbikes *and*

station. *The node labelled with* station *is on the path* P. *As shown in Figure 4.1a, there are two nodes that are labelled with* station *under* Dublin *and they are on the same path. Additionally, those two* station *nodes are reachable from the node* Dublin. ■

As depicted in this section, XML data has a tree-based representation. In the next section, we introduce an XML query language, which navigates the XML tree and locates nodes from it, and the corresponding XML query data model.

4.2 XML Query Language

In this dissertation, we are concerned with a subset of XPath expressions, which is used to navigate and locate instance nodes within an XML tree, and it is a rather robust subset of XPath: many applications use only expressions in this subset and it is also an important component (sub-expression) of XQuery expressions that are commonly used to query a broad spectrum of XML information sources [ZLBT03]. The subset of XPath expressions is specified by the following grammar:

Expr	$expr ::= /q \mid q$
Path	$q ::= q_1/q_2 \mid q_1[q_2] \mid \alpha : n$
Axis	$\alpha ::= child \mid descendant \mid attribute$
NameTest	n, where $n \in \Sigma$

In the above grammar, the three options of *Axis* can be abbreviated to "/", "//" and "@", respectively. The single slash "/" represents the parent-child relationship, the double slash "//" means the ancestor-descendant relationship and "@" indicates the attribute. An expression can be expressed as one or more paths. Each path contains an axis and a NameTest. The combination of a single axis and NameTest is also referred to as a *step*. An expression that is included in a pair of square brackets is called a *predicate*. Each step has a context (see §4.2.2), which is referred to as the intermediate result that is produced by the previous step. Two sample XPath expressions are given in Example 4.2 and Example 4.3.

Example 4.2 (A Single Path Expression)

Expression: //bikes/city//Brisbane//station

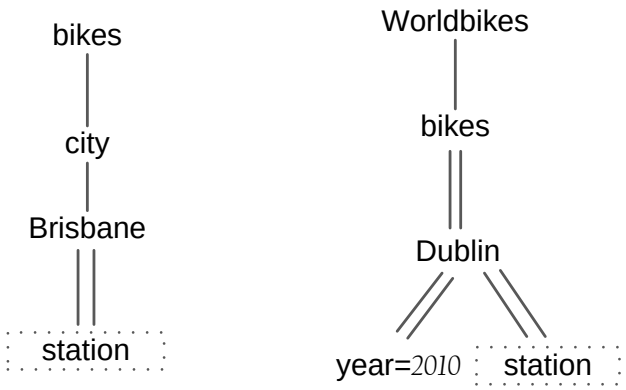
Description: list all information of bicycle stations in Brisbane ■

Example 4.3 (An XPath Expression with Predicate)

Expression: //Worldbikes/bikes//Dublin[//year=2010]/station

Description: list all information of bicycle stations in Dublin in 2010 ■

The two sample XPath expressions listed in Example 4.2 and Example 4.3 are based on the Worldbikes dataset. An XPath expression with no predicate is referred to as a *single path expression* (Example 4.2). When evaluating an XPath expression, the intermediate result (a sequence of instance nodes) produced by each *path* expression is passed as the input to the evaluation of the next path expression (see §4.2.2 for more detailed discussion). The result of the last path expression that is not a predicate serves as the result of the entire XPath expression, e.g., //station of the single path expression in Example 4.2.



(a) Tree Pattern 1 **(b)** Tree Pattern 2

Figure 4.2: Tree Patterns Correspond to the XPath Expressions in Example 4.2 and 4.3

4.2.1 Tree Patterns

The subset of the XPath expression mentioned previously can be expressed by a more general representation named *tree pattern queries* or simply *tree pattern*s. A tree pattern contains only parent-child and ancestor-descendant relationships and it has an *arity* [MS02]

assigned to it indicating the number of *return/output node*s in the tree pattern. The subset of the XPath expression introduced in last section is a special case of the tree pattern with arity 1, that is, only one output node is defined. A tree pattern is denoted by p and is represented by a 4-tuple as shown in Definition 4.3.

Definition 4.3 [Tree Pattern]
Given a tree pattern p, p is represented by a 4-tuple $<\mathbb{R}_p, \mathbb{N}_p, \mathbb{E}_p, \mathbb{L}_p>$, where $\mathbb{R}_p$ is the root of the tree pattern, $\mathbb{N}_p$ is a set of nodes in p, $\mathbb{E}_p$ is a set of edges containing only / (parent-child relationship) and // (ancestor-descendant relationship) and $\mathbb{L}_p$ is a set of labels, where $\mathbb{L}_p \subseteq \Sigma$. ■

The tree patterns in Figure 4.2 correspond to the XPath expressions in Example 4.2 and 4.3. Edges between nodes represent the corresponding relationships (axes) between them. The root node has an implicit ancestor-descendant relationship to the document root. Nodes that are surrounded with dotted rectangular boxes are the return nodes of the tree patterns. Tree patterns and the SchemaGuide which is introduced in the next chapter, are essential concepts of our containment checking algorithm (see Chapter 5).

4.2.2 Sequences and Instances

By definition [W3C10c], during the evaluation of an XPath expression, each path expression is passed a *sequence* of nodes from the previous path expression and generates a new sequence that serves as the input to the next path expression. A sequence with one node (document element) is passed as the input for the initial path expression. When a predicate is encountered, it is applied to the node sequence and filtered based on the predicate. Eventually, when the last non-predicate path is reached, a *serialisation* process is performed to return all subtree nodes of each node in the sequence.
Therefore, based on the above evaluation process, we define the core component of our XPath data model as *instance node* and *sequence*, where an *instance node* represents a node in an XML tree and a *sequence* is a list of instance nodes in document order. As will be shown in §4.3.1, a set of algebraic operators, which operate on the XPath data model, are defined according to the semantics of each evaluation step.

To keep our data model abstract so as to support different physical data models and storage techniques, an instance node is not required to carry references to their parent or child in the XML tree. This information may be explicitly included through concrete implementations. However, for the purpose of our model, all operations are based upon sequences and instance nodes.

4.3 XFM View Model

In this section, we now introduce our XFM view model. As part of this discussion, it is necessary to describe our algebraic operators as they form part of the view graph [LRB10b]. We then show how XPath expressions are translated into our data model.

4.3.1 Sequence-Based Algebraic Operators

We now introduce our sequence-based algebraic operators. Our main objective is to provide a view model using the algebraic operators and then build our fragment-based view framework based on this view model. There are three types of operators, *dependency join*, *select* and *deep project* operator. We give a detailed description for each of them as listed below.

Dependency Join Operator (*djoin*, $\overset{\alpha}{\underset{\rightarrow}{\bowtie}}$). As explained previously, an XPath expression is comprised of a sequence of *path* expressions and this sequence of expressions is represented as a chain of *dependency join*s (*djoin*). The input of the djoin operation are two sequences of instance nodes: 1) a sequence of instance nodes resulting from the previous evaluation and 2) another sequence of instance nodes satisfying the NameTest specified in the context path expression. The output of the djoin operation is a sequence of instance nodes resulting from the axis operation and predicate filtering. We refer to the input sequence as *context sequence* since it serves as the *context* to the current operation.

Definition 4.4 [Semantic of a `djoin` Operation]
A djoin *operation is a binary operation written as* $S \overset{\alpha}{\underset{\rightarrow}{\bowtie}} S'$*, where both* S *and* S' *are sequences of instance nodes and* α *is an axis. The result of a djoin operation is a sequence*

of instance nodes which fulfills the dependency condition (axis α). The semantics of a dependency join is provided as:

$$S \underset{\rightarrow}{\overset{\alpha}{\bowtie}} S' = \bigoplus_{i=0}^{k} v_i, v_i \in S', 0 \leq k \leq |S'|$$

$$\bigwedge \exists u \in S, (u, v_i) \rightarrow \begin{array}{l} \text{PC} \bigwedge (u, v_i) \in \mathbb{E}_t, \textit{ if } \alpha \rightarrow \text{PC} \\ \text{AD} \bigwedge (u, v_i) \in \mathbb{E}_t^+, \textit{ if } \alpha \rightarrow \text{AD} \end{array}$$

where $\bigoplus$ is the concatenation function, which creates a sequence by concatenating instance nodes (v_i) satisfying the dependency condition specified by α, either PC *(parent-child) or* AD *(ancestor-descendant).* ■

Select Operator (*select*, σ_{pred}). Recall that a *path* expression may contain an optional set of predicates. For each predicate, the select operation performs a filtering over the context sequence, which selects instance nodes satisfying the predicate.

Definition 4.5 [Semantic of a `select` Operation]
A select *operation is an unary operation written as $\sigma_{pred}(S)$ where* pred *is a condition of the selection and S is the context sequence. This operation selects a sequence of instance nodes in S for which* pred *holds. The semantics of the* select *is:*

$$\sigma_{pred}(S) = \bigoplus_{i=0}^{k} v_i, v_i \in S, 0 \leq k \leq |S| \bigwedge v_i \rightarrow pred$$

The result of a *select* operation is a sequence of instance nodes, which is created by concatenating all instance nodes that satisfy the predicate.

Deep Project Operator (*dproject*, $\Pi^D(S)$). A Deep Project operation returns a specified sequence of instance nodes together with all instance nodes within their subtree. When an XPath expression generates its final set of result nodes, it will always return the entire subtree beneath each instance node according to the W3C XPath Recommendation [W3C10a]. For this reason, we define a *deep project operation* (*dproject*) to project the *entire* subtree content of each instance node in the sequence.

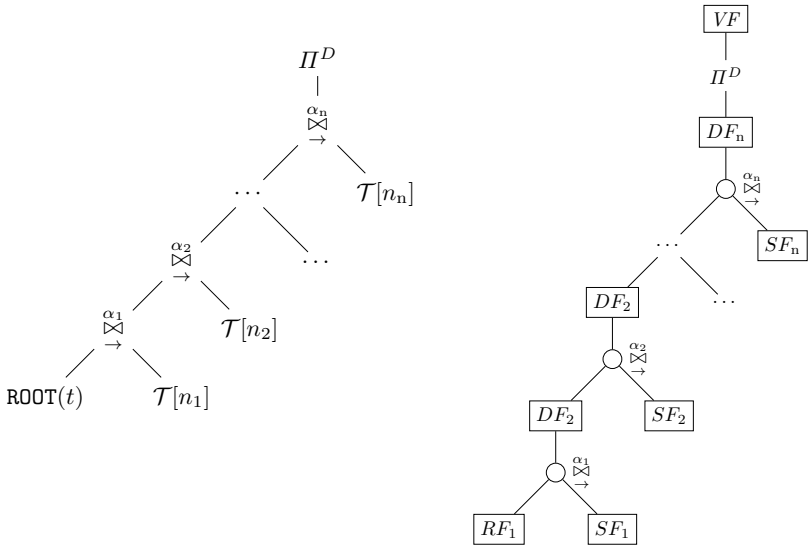
Definition 4.6 [Semantic of a `dproject` Operation]

A deep project *operation is a unary operation written as* $\Pi^D(S)$*. This operation projects the entire subtree content of instance nodes within* S *resulting a sequence of subtrees, where each subtree is represented by a sequence of instance nodes within the subtrees. The semantic of the* dproject *is:*

$$\Pi^D(S) \quad = \quad \bigcup_{i=0}^{k} t_{v_i},\ v_i \in S,\ 0 \leq k \leq |S| \bigwedge t_{v_i} \in t$$

where t_{v_i} *is a subtree in* t*, which rooted at node* v_i*.* ■

The result of a *dproject* is a union of subtrees that are rooted at instance nodes within the context sequence.

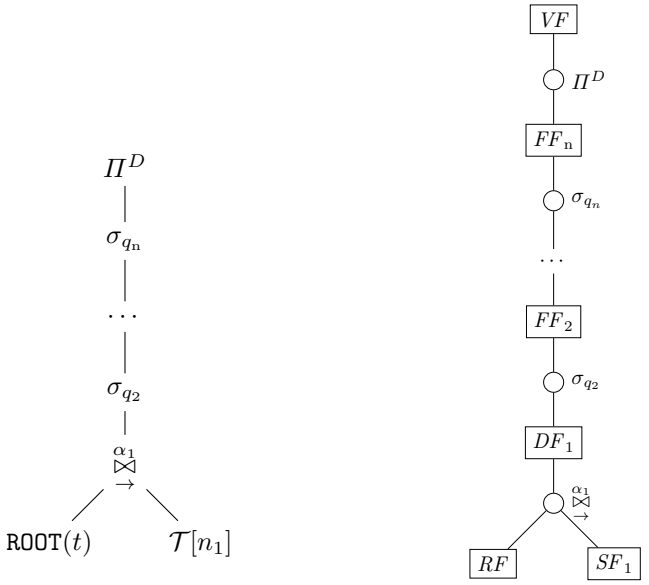


(a) Algebra Representation (*Case* 1) **(b)** XFM View Graph

Figure 4.3: Algebra and Fragment-Based Representation (*Case* 1)

4.3.2 From XPath to Algebraic Representation

In this part, we demonstrate how to translate XPath expressions into our view model based on the algebraic operators just introduced. In our XPath data model, an expression e is



(a) Algebra Representation (*Case* 2)

(b) XFM View Graph

Figure 4.4: Algebra and Fragment-Based Representation (*Case* 2)

a sequence of instance nodes resulting from evaluating e over an XML document. We define a translation function $\mathcal{T}$ and the translation of an expression e into the algebraic representation is denoted by $\mathcal{T}[e]$. For each XPath expression, we add an *initial execution context* to it, denoted by ROOT(t), which represents a sequence with only one instance node, the root of the XML tree. Based on the subset of the XPath expressions, there are two cases when translating XPath expressions into the algebraic representations. *Case 1* summarises expressions with no predicate and *Case 2* outlines the circumstance that expressions may contain predicates.

Case 1 When e = q_1/q_2/q_3 ... q_n with no predicate, where q_i = α_i/n_i and $1 \leq i \leq \mathrm{n}$ (see Figure 4.3a):

$$\mathcal{T}[e] = \Pi^D(((\texttt{ROOT}(t) \underset{\rightarrow}{\overset{\alpha_1}{\bowtie}} \mathcal{T}[n_1]) \underset{\rightarrow}{\overset{\alpha_2}{\bowtie}} \mathcal{T}[n_2]) \ \ldots \underset{\rightarrow}{\overset{\alpha_\mathrm{n}}{\bowtie}} \mathcal{T}[n_\mathrm{n}]);$$

Case 2 When e = $q_1[q_2][q_3]...[q_n]$ (see Figure 4.4a):

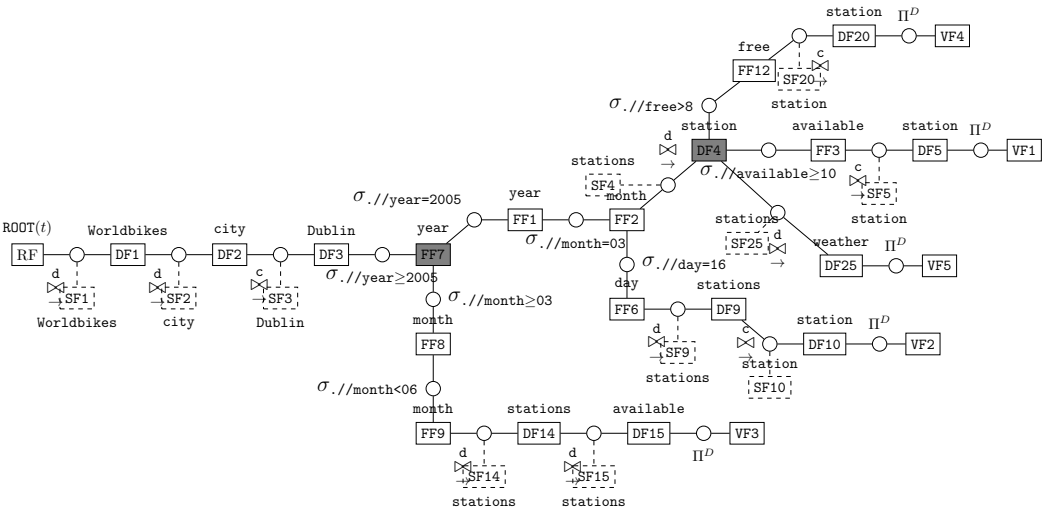
$$\mathcal{T}[e] = \Pi^D(\sigma_{q_\mathrm{n}}(\ldots\sigma_{q_3}(\sigma_{q_2}(\texttt{ROOT}(t) \underset{\rightarrow}{\overset{\alpha_1}{\bowtie}} \mathcal{T}[q_1]))));$$

As shown in *Case* 1, path expressions are connected by a chain of djoin operators starting from the initial context ROOT(t). After all djoin operations are executed, a dproject operator is then applied to the sequence of instance nodes produced by the last path expression. Whereas, for *Case* 2, a djoin operation is first executed upon q_1 and ROOT(t). The select operations are performed sequentially over the sequence of instance nodes produced by the initial djoin operation. As with *Case* 1, a dproject operation is finally applied to the sequence of instance nodes resulting from the last select operation.

As will be shown in §4.4, the algebraic representation is a fundamental construct for the XFM view graph.

4.4 XFM Fragments and View Graph

In this section, we present the XFM view graph, which is built on top of the XFM view model and a classification of fragments as discussed in the §4.4.1.



V_1 : List operation status of stations that have more than 10 bikes available in Dublin in March 2005.
//Worldbikes//city/Dublin[.//year=2005][.//month=03]//stations[.//available>10]/station

V_2 : List operation status of each station in Dublin on the 16^{th} of March 2005.
//Worldbikes//city/Dublin[.//year=2005][.//month=03][.//day=16]/stations/station

V_3 : List bike availability of each station in Dublin for the months of March, April and May after 2005.
//Worldbikes//city/Dublin[.//year≥2005][.//month≥03][.//month<06]/stations//available

V_4 : List stations in Dublin that always have more than 8 free bicycle stands in March 2005.
//Worldbikes//city/Dublin[.//year=2005][.//month=03]//station[.//free>8]/station

V_5 : List weather conditions at all stations in Dublin in March 2005.
//Worldbikes//city/Dublin[.//year=2005][.//month=03]//stations//weather

Figure 4.5: XML Fragment Materialization View Graph

4.4.1 View Fragments

As outlined previously, the XFM view model is based on a set of algebraic operators where for each of these operators, we define different types of *fragment*s representing the result of the algebraic operators. Basically, a fragment (except *Source Fragment*) is a sequence of instance nodes resulting from an algebraic operation. As shown in §4.4.2, it is these fragments that can be shared across XML views and when applied to the XFM view model, forms our fragment-based view framework. A sample graph is shown in Figure 4.5, with fragments represented as rectangular boxes. Fragments are categorised into 5 types as listed below:

- **Root Fragment (RF)** – The RF fragment is a 4-ary tuple <*fid*, $\mathbb{V}$, n, p >, where *fid* is the unique identifier assigned to RF; $\mathbb{V}$ is a set of views sharing this fragment; n represents the *NameTest* and the label (tag name) of the *return node* in p; and p is the tree pattern mapped to RF from which the RF fragment can be evaluated. For a Root Fragment, p is simply a tree with a single node represented by the document node. The RF fragment represents the *initial execution context*. It contains a sequence with only one instance node, which is the root of an XML tree (also known as the document node). It always represents the starting point of an XFM view graph. While a view graph will contain multiple query representations, they are all joined by the same root fragment, as shown in Figure 4.5, e.g., RF with rectangle box.

- **Filter Fragment (FF)** – The FF fragment is a 5-ary tuple <*fid*, $\mathbb{V}$, n, *pred*, *p*>, where *fid* is the unique identifier assigned to FF; $\mathbb{V}$ is a set of views sharing this fragment; n is the *NameTest* as well as the label (tag name) of the *return node* in p; *pred* is the predicate applied to the context sequence; and *p* is a tree pattern mapped to the FF fragment from which the FF fragment can be evaluated (p is a subpart of the original query). The FF fragment (e.g., FF1 in Figure 4.5) represents the result produced by a *select* operation. In our view model, the select operation always contains a predicate used to filter an input sequence, e.g., *free>8* in Figure 4.5 represents the filter operation that results in FF12.

- **Dependency Join Fragment (DF)** – The DF fragment is a 6-ary tuple <*fid*, $\mathbb{V}$, n_{left},

n_{right}, α, *p*>, where *fid* is the unique identifier assigned to DF; $\mathbb{V}$ is a set of views sharing this fragment; n_{left} is the *NameTest* of the left join operand and the label of (tag name) the *return node* in p; n_{right} is the *NameTest* of the right join operand; α is the axis of *djoin* (the relationship between two operands) operation; and *p* is the tree pattern from which a DF fragment can be evaluated, (p is a subpart of the original query). A DF Fragment (e.g., DF1 in Figure 4.5) represents the sequence resulting from a *djoin* operation, e.g., the $\overset{d}{\underset{\rightarrow}{\bowtie}}$ before DF1 represents a dependency join operation, where d indicates the ancestor-descendant relationship.

- **Source Fragment (SF)** – The SF fragment is a triple <*fid*, $\mathbb{V}$, n >, where *fid* is the unique identifier assigned to SF; $\mathbb{V}$ is a set of views sharing this fragment; and n represents the *NameTest*. The SF fragment represents the full set of instance nodes whose labels match n. The major difference between this fragment and all others is that it cannot be reused and merely acts as an operand in a *djoin* operation.

- **View Fragment (VF)** – A VF fragment is a 4-ary tuple <*fid*, $\mathbb{V}$, n, *p*>, where *fid* is the unique identifier assigned to VF; $\mathbb{V}$ is a set of views sharing this fragment; n is the *NameTest* and the label (tag name) of the *return node* in p, and p is a tree pattern from which a VF fragment can be evaluated (in this case, p equals to the entire query). A VF fragment (e.g., VF1 in Figure 4.5) represents the final result for the view definition. It always follows a deep project operation, e.g., Π^D before VF1 indicates the deep projection.

4.4.2 XFM View Graph

The XFM View graph is a directed acyclic graph consisting of two types of nodes: a *fragment node* denoted by f, and an *operation node* denoted by o. Fragment nodes can have operation nodes only as neighbours and operation nodes can have only fragment nodes as neighbours. The Root Fragment is the root of the XFM view graph. A formal definition of an XFM View Graph is given in Definition 4.7.

Definition 4.7 [XFM View Graph]

An XFM View Graph is a directed acyclic graph represented by a 4*-tuple, denoted by* $\mathcal{G}$ = $< f_r, \mathbb{F}, \mathbb{O}, \mathbb{V} >$*, where* f_r *is the RF fragment,* $\mathbb{F}$ *is a set of* F-Node*s,* $\mathbb{O}$ *is a set of* O-Node*s*

and $\mathbb{V}$ *is a set of views in* $\mathcal{G}$. ■

A sample XFM view graph is shown in Figure 4.5, where fragment nodes are represented by rectangular boxes and operation nodes by circles, and a list of natural language descriptions of the views appearing in the sample graph and the corresponding XPath expressions. Within an XFM view graph, a *simple path* determines a path that has distinct edges in a view graph. A simple path starting at the RF fragment and ending at the VF fragment, uniquely identifies an XPath expression/view in an XFM view graph. We use the notation V to represent the view that is uniquely identified by a simple path. We assign a level value to each fragment with the exception of SF fragments, within the XFM view graph. As shown in Figure 4.5, the XFM graph contains five views V_1 to V_5 that end with VF fragments, VF1, VF2, . . ., VF5, respectively. The fragments in gray, FF7 and DF4 are the only materialised fragments in this graph. Bear in mind that we use it only for the purpose of illustration as the selection algorithm may choose different fragments for materialisation. In the rest of this dissertation, we use fragment and fragment node interchangeably and also algebraic operator and operation node.

4.5 Constructing the View Graph

At the end of §4.3.2, we have shown view (XPath) expression in algebraic format. In §4.4, we described the view graph, components and operators. The final step is relatively simple: transforming the algebraic view into the graph representation, and merging views into a global XFM view graph. We begin with a description of the transformation process as listed below:

1. The initial execution context `ROOT`(t) is transformed to a *fragment node* representing the RF fragment.

2. Algebraic operators are transformed into *operation nodes*.

3. Depending on the type of the algebraic operation encountered, i.e., djoin, selection or dproject, the transformation process generates different types of fragments indicating the result of the algebraic operations and those fragments are transformed to the *fragment nodes* and appended after the corresponding operation node.

4. During the process depicted in Step 3, if an *operation node* implies a *djoin* operation, then a SF fragment is also generated representing the right operand of the djoin operation and is then transformed into a *fragment node*.

Figure 4.3b and 4.4b are the XFM view graphs representing single XPath views after performing the transformation process on the algebraic representation shown in Figure 4.3a and 4.4a.

4.5.1 The XFM View Graph Construction

The approach taken in the construction process is to first transform all views into the corresponding graph representations, which includes *XPath to Algebraic Representation Translation* and *Algebraic Representation to XFM View Graph Transformation*, and then merge all those single view based XFM view graphs together.

Algorithm 4.1: XFMViewGraphConstruction(E)

Input: A set of XPath expressions $E = \{e_1, e_2 \cdots e_n\}$
Output: An XFM view graph $\mathcal{G}$

```
1  Transform e1, e2 ··· en into the graph representation, V1, V2 ··· Vn, respectively;
2  set G to be an empty graph;
3  foreach Vi ∈ {V1, V2 ··· Vn} do
       /* Case 1:  When G is empty                                   */
4      if G is empty then
5          add Vi into G;
       /* Case 2:  When there is one or more views in G              */
6      else
7          foreach f' in Vi do
8              if there exits a fragment in G that is equivalent to f' then continue;
9              else
10                 search for a fragment f in G after where f' is inserted;
11                 TargetViewAdaptationForInsertFragment(f, f', Vi,
                   G);
```

As shown in Algorithm 4.1, a set of XPath expressions/views are accepted as the input. During the initialisation, all XPath expressions are transformed into the graph representation (*Line 1*) and an empty XFM view graph is created (*Line 2*). The graph construction algorithm is a special case of the view adaptation process, where a new view is treated as a list of linked fragments starting from the RF fragment representing the document. The

merging of the view graph is achieved by inserting the fragments of the new view into the global XFM view graph. By iteratively verifying the containment relationship between the new fragments and existing ones, the algorithm determines the place in the XFM view graph where the insertion should take place. This description is deliberately brief for now as we discuss containment checking in depth, in Chapter 5.

In summary, there are two possible cases, as outlined in Algorithm 4.1, when adding a view V_i into the global XFM view graph $\mathcal{G}$,

Case 1 When $\mathcal{G}$ is empty. This case is managed by directly adding V_i into $\mathcal{G}$ by setting the root of $\mathcal{G}$ as the root of V_i, *Line 5-6*.

Case 2 When $\mathcal{G}$ contains one or more views. The algorithm checks for each fragment f' in V_i: whether f' is equivalent to an existing fragment. If not, then f' is added into the global view graph by calling the `TargetViewAdaptationForInsertFragment` function, *Line 8-12*.

`TargetViewAdaptationForInsertFragment` (see Algorithm 7.2 on Page 113) is discussed in detailed in Chapter 7. For the moment, one needs only know that `TargetViewAdaptationForInsertFragment` accepts two fragments, an existing one (i.e., f) from $\mathcal{G}$, and a new one (i.e., f') from V_i to be inserted into $\mathcal{G}$.

4.6 Summary

In this chapter, the XML data model and query language were introduced, followed by a description of XML queries and XPath processing. We then provided a detailed discussion on the XFM view model and graph. At this point, we can represent XPath views in our system and are ready to consider view management, including containment checking, fragment selection and adaptation which form the basis of the next three chapters.

Chapter 5

Containment Checking

In this chapter, we present our containment checking mechanism, which is based on a new metadata structure, the *SchemaGuide*. The containment checking process is necessary to identify common parts between views and to determine the extent of changes between new and old view definitions. The chapter begins with a strategy overview in §5.1 and then in §5.2, we present the SchemaGuide construct which is the key component in our containment checking algorithm. In §5.3, the concepts of *Embedding* and *Embedded Trees* are introduced, before our containment algorithms are discussed in §5.4.

5.1 Strategy Overview

When a view definition has changed, it requires the comparison between old and new definitions, with changes identified at the fragment level. Therefore, our containment mechanism focuses on a fragment comparison process and in particular, whether or not the new fragment is contained within the old fragment. View adaptation is rarely confined to the modification, addition or deletion of a single fragment as it often impacts on connected fragments in the XFM view graph.

As summarised in [Sch04], the most general notion of containment is based on the evaluation result of two tree patterns (see Chapter 4), to determine if the result of a tree pattern is contained in another one. Based on this general notion, our containment checking algorithm checks whether the results of two tree patterns are contained. We differ however, in that we manage containment based on the sets of matched *subtrees* retrieved from a SchemaGuide,

that match the tree patterns. Our approach provides both accuracy and improved performance as shown in [LRB10a, LRB11].

Since a view is comprised of a number of fragments, a fragment is always represented by a sub-query and we use the concept of a tree pattern to represent this sub-query. Thus, our containment checking algorithm evaluates whether one tree pattern is contained in a second tree pattern.

In summary, the containment check involves a single step which compares two tree patterns and evaluates to *true* or *false*. For the purpose of explanation, we separate our discussion into two streams: the first involving a structural check between tree patterns and the second focusing on value-based predicates. The structural check evaluates the containment relationship between two tree patterns using the SchemaGuide. Both structure (pattern and relationship between nodes) and data (label name) must be preserved between the two tree patterns. The second discussion takes predicates into consideration. We begin our discussion with the introduction of the SchemaGuide that is used extensively by the containment checking algorithm.

5.2 The SchemaGuide

A *SchemaGuide* [LRB10a] is a tree-based metadata structure that summarises the structural (pattern) information of an XML tree and describes constraints. It is used by the containment checking algorithm to determine the containment relationship between fragments and identify the common sub-expressions between views. Moreover, as will be shown in Chapter 7, it provides metadata that can be exploited for performance gains.

A *SchemaGuide* describes the structure of an XML tree without concern for the content of the corresponding XML document. Similar to other existing metadata structures, it also provides constraints on XML data that are either explicitly defined in a DTD file or an XML schema, or implicitly outlined by a structural-based summary such as a strong DataGuide [GW97] or a QueryGuide [IHH09]. The difference between the SchemaGuide and existing metadata structures is that it provides more detailed structural information of XML documents, e.g., the subtree structure of each instance node and the order of each instance node appearing in the subtree. Similar to an XML tree, a SchemaGuide has a tree based structure

and is represented by a 4-tuple as depicted in Definition 5.1.

Definition 5.1 [SchemaGuide]

Given an XML document and its tree representation t, a SchemaGuide G is a tree summarising the structural information in t and it is represented by a 4-tuple $< \mathbb{R}_G, \mathbb{N}_G, \mathbb{E}_G, \mathbb{L}_G >$ where $\mathbb{R}_G$ denotes the root node of G; $\mathbb{N}_G$ is a set of nodes in G; $\mathbb{E}_G$ represents a set of edges in G; and $\mathbb{L}_G$ contains a set of labels in G, where $\mathbb{L}_G \subseteq \Sigma$. ■

If G is a SchemaGuide corresponding to an XML tree t, then we say that G *summarises* t or t *conforms to* G, denoted by $G \vDash t$. Given a schema node u, where $u \in \mathbb{N}_G$, $\mathbb{L}_G(u)$ returns the label of u and $\mathbb{L}_G(u) \in \Sigma$. $\mathbb{E}_G^+$ is the transitive closure of $\mathbb{E}_G$ defining the ancestor-descendant relationship between any arbitrary pair of schema nodes in G.

Figure 4.1b (see Page 45) is a snapshot of the SchemaGuide corresponding to the segment of the *Worldbikes* dataset shown in Figure 4.1a. Each instance node of an XML tree is mapped to a node within the SchemaGuide, to differentiate between nodes in an XML tree and nodes in a SchemaGuide, we refer to nodes in a SchemaGuide as the *schema nodes*. Each schema node is uniquely identified by an integer value called the ***schema node id*** (sid), e.g., the number that is associated with each node in Figure 4.1b. We use g_u to denote a subtree of a *SchemaGuide* rooted at the schema node u and it contains all nodes that are transitively *reachable* from u, e.g., a subtree rooted at *time* in Figure 4.1b is denoted by g_{time}.

As depicted in Definition 5.2, for an XML tree and the corresponding SchemaGuide, we define a mapping function, φ, which maps all nodes between the XML tree and the corresponding SchemaGuide. The mapping function must retain all characteristics as specified in Definition 5.2 and serves as an essential concept for our later propositions and proofs. It is the many-to-one characteristic of the mapping function that makes it suitable for an XML tree to SchemaGuide mapping where there are generally many instances of a schema node in the XML tree.

Definition 5.2 [The Mapping Function (φ)]

Given a tree t and a SchemaGuide G, $G \vDash t$. Let $\varphi : \mathbb{N}_t \rightarrow \mathbb{N}_G$ be a mapping function where the following characterises are preserved:

1. *Root preserving:* $\varphi(\mathbb{R}_t) = \mathbb{R}_G$.

2. *Edge preserving: if* $(u, v) \in \mathbb{E}_t \longrightarrow (\varphi(u), \varphi(v)) \in \mathbb{E}_G$.

3. *Label preserving:* $\forall\ u \in \mathbb{N}_t \longrightarrow \mathbb{L}_t(u) = \mathbb{L}_G(\varphi(u))$.

4. *Root-to-Node Path preserving:* $\forall\ u \in \mathbb{N}_t$ *and* u *is reachable from* $\mathbb{R}_t$ *by the path* $P \longrightarrow \varphi(u)$ *is reachable from* $\varphi(\mathbb{R}_t)$ *by the same path* P.

5. *Subtree Structure preserving:* $\forall\ u \in \mathbb{N}_t \bigwedge \forall\ v \in \mathbb{N}_{t_u} \longrightarrow \varphi(v) \in \mathbb{N}_{G_{\varphi(u)}} \bigwedge \mathbb{L}_t(v) = \mathbb{L}_G(\varphi(v))$.

6. *Order preserving:* $\forall\ u \in \mathbb{N}_t \bigwedge \forall\ v_i \in \mathbb{N}_{t_u}$, $0 < i < k$, *where* k *is the number of children of* $u \longrightarrow \varphi(v_i) \in \mathbb{N}_{G_{\varphi(u)}}$ *and* $\varphi(v_i)$ *is the* i*th child of* $\varphi(u)$. ■

As shown in Definition 5.2, when mapping nodes from an XML tree t to a SchemaGuide G, the mapping function φ guarantees: 1) the root node between t and G are mapped; 2) for any pair of instance nodes in t, their edge must be identical to the edge between their mapped schema nodes in G; 3) the same labels are assigned to nodes in t and G that are mapped; 4) for any instance node in t and its mapped schema node in G, they are on the same path; 5) for any mapped nodes in t and G, they must have identical subtree structure and; 6) the order of nodes within the subtree must remain same. Existing research has focused on the first four characteristics and we extend existing work by providing the more detailed metadata information required by 5 and 6. This is the key contribution in our SchemaGuide as it made possible to reduce the search space required for containment checking.

Based on the mapping function defined in Definition 5.2, we now present a new property for our SchemaGuide. Property 5.1 outlines the fact that if an XML tree t conforms to a SchemaGuide G, then there must exist a many-to-one mapping between instance nodes in t and schema nodes in G. Every instance node in t maps to a schema node in G, whereas, every schema node in G can be mapped to at least one instance node in t.

Property 5.1 *Given an XML tree* t *and a SchemaGuide* G*, if* $G \models t$*, then there exists a mapping function* φ *which maps every instance node in* t *to a schema node in* G*, whereas, a schema node can be mapped to multiple instance nodes.* ■

Furthermore, according to Property 5.1, each instance node within an XML tree maps to a schema node in the corresponding SchemaGuide and as a result, it derives the corresponding *sid* from the mapped schema node. As was shown in Figure 4.1a, each instance node is associated with an integer value representing the sid. Based on the concepts outlined in Definition 5.2 and Property 5.1, for any XML tree, one could build its corresponding SchemaGuide during the XML document parsing process by temporarily storing all root-to-leaf and subtree structures of each instance node. A schema node is only created if the root-to-leaf and subtree structure associated with an instance node have never been previously encountered during the parsing. The algorithm for generating a SchemaGuide were implemented using the Xerces2 SAXParser [Xerces2], which scans an XML document and each time it encounters a tag, it calls the corresponding tag handler method. We record all paths and subtree structures by iteratively concatenating/substracting tags that are encountered during the parsing.

5.3 Embedding and Embedded Trees

In this section, we introduce the concepts that are necessary for our containment checking algorithm. Recall that a tree pattern is a subset of an XPath expression. An *embedding* function maps the tree pattern to an instance in an XML tree. To return all instances that match a tree pattern, multiple embeddings are required. This is formalised in Definition 5.3 which outlines that for each tree pattern p, there exists one or more subtrees in t that maps p to t with structural characteristics preserved. This provides a platform for evaluating containment at schema level. As mentioned in Chapter 4, a subtree of a tree t is a tree consisting of a node in t and **all** or **part** of its descendants. A subtree rooted at a node u in t is denoted by t_u. Also, a subtree without context information, i.e., the root of its original tree, is itself a tree. From now on, we use the term tree and subtree interchangeably.

Definition 5.3 [XML Tree Embedding]
Given a tree pattern p and an XML tree t, let $e : \mathbb{N}_p \rightarrow \mathbb{N}_t$ be an embedding of p in t where $\mathbb{E}_p = \mathbb{E}_p^{/} \cup \mathbb{E}_p^{//}$, $\mathbb{E}_p^{/}$ and $\mathbb{E}_p^{//}$ contain edges indicating all parent-child and ancestor-descendant relationships, respectively, and the following characteristics are preserved:

- *Root preserving: $e(\mathbb{R}_p) = \mathbb{R}_t$*

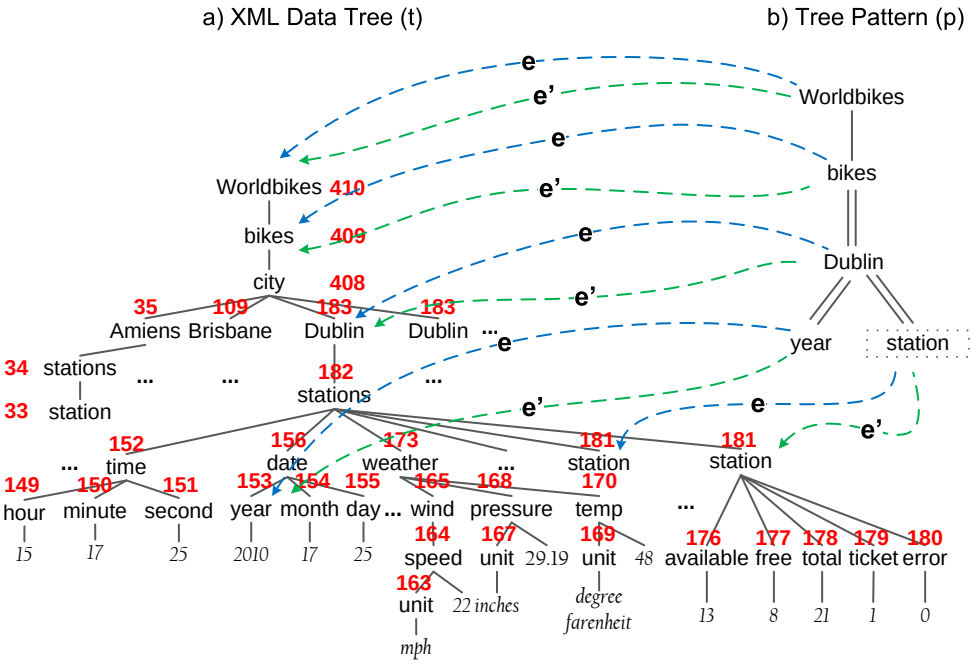


Figure 5.1: XML Tree Embedding

- *Child edge preserving: if* $(u, v) \in \mathbb{E}_p^{/} \longrightarrow (e(u), e(v)) \in \mathbb{E}_t$
- *Descendant edge preserving: if* $(u, v) \in \mathbb{E}_p^{//} \longrightarrow (e(u), e(v)) \in \mathbb{E}_t^{+}$
- *Label preserving:* $\forall u \in \mathbb{N}_p \longrightarrow \mathbb{L}_p(u) = \mathbb{L}_t(e(u))$ ■

A tree pattern contains parent-child (child axis) and ancestor-descendant (descendant axis) relationships, while ancestor and parent axes can be handled by translating them into the corresponding child and descendant axes [OMFB02]. Definition 5.3 states that the embedding process requires that the root of p maps to the root of t_u, where u is a node in t and u is the embedding of $\mathbb{R}_p$ in t; all parent-child and ancestor-descendant relationships are preserved; and matching nodes in p and t_u have identical labels.

Example 5.1 (XML Tree Embedding in Figure 5.1)

There are two embeddings (e and e') between the tree pattern p and the XML tree t. Both e and e' map the root node of p to the root node of t (root preserving). The label of each pair of mapped nodes are identical (label preserving), e.g., node labelled with bikes *in p is mapped to the node labelled with* bikes *in t by both e and e'. Edges (relationships) between tree*

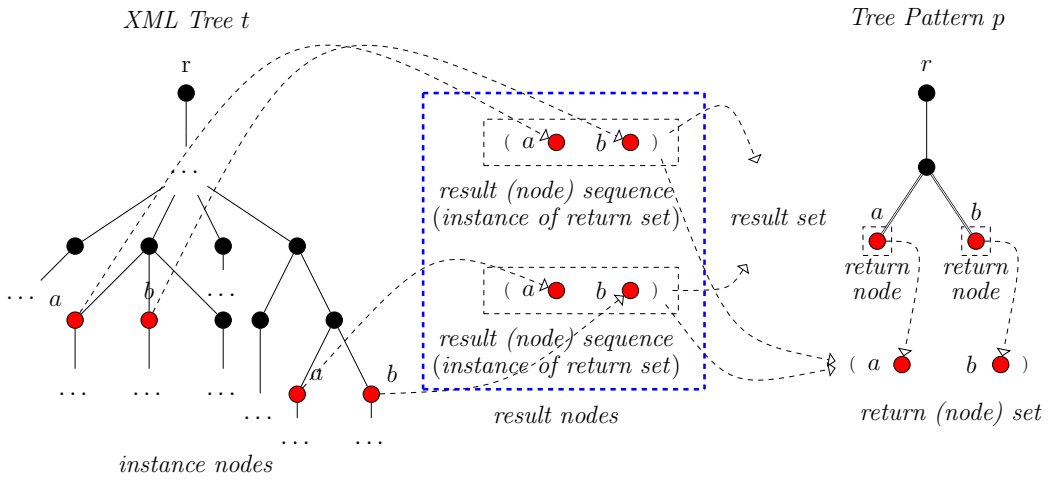


Figure 5.2: XML Tree and Tree Pattern

pattern nodes are consistent with the corresponding edges between the mapped instance nodes, e.g., both Dublin *and* year *in* p *and* t *have ancestor-descendant relationships (child and descendant edge preserving).* ■

Before we outline our discussion, some terminologies are introduced now to help understand definitions and propositions for our containment checking mechanism. Recall that a tree pattern may have multiple *return node*s. As shown in Figure 5.2, the nodes a and b are the *return nodes* of the tree pattern p and the set containing both a and b is referred to as the *return node set* of p or simply the *return set*, and nodes within the XML tree t are *instance nodes*. When evaluating p over t, all instance nodes match to the return nodes are returned, we refer to those instance nodes as the *result nodes* of the tree pattern p. Among all result nodes, there are several sequences which group a set of result nodes together and map to the *return set* (they are instances of the return set), we refer to them as the *result node sequence*, or simply *result sequence*. A set of all potential *result sequences* are referred to as the *result set*.

We now define the evaluation result of executing a tree pattern p over an XML tree t. Based on the embedding function in Definition 5.3, the following equation formally defines the results of evaluating a tree pattern p over an XML tree t, which is a set of sequences of

instance nodes that are extracted from each embedding,

$$p(t) = \{e(\overline{u}) \mid e \text{ is an embedding of } p \text{ in } t, \\ \overline{u} = (u_1, u_2, \ldots, u_k) \text{ is the } \textit{return set} \text{ of } p\} \quad (5.1)$$

The above equation shows that the evaluation of a tree pattern over an XML tree is a set of k-tuples derived from the XML tree embedding, where k is the arity of the tree pattern and every k-tuple represents a *result sequence* containing instance nodes in the XML tree.

Example 5.2 (Evaluating p in Figure 5.3 where *station* is the return node)
The result nodes for p are determined through all possible embeddings of p in t. In this case, station *is the return node of p and the result of evaluating p is all* station *nodes resulting from the embeddings. In the sample XML tree, two* station *nodes associated with sid 181 are the result of p. They both have the same sid as they map to the same schema node. Note that the node station with sid 33 does not preserve the root-to-node-path (e.g., city-Dublin-station and city-Amiens-station).* ■

The matched instances of a tree pattern returned by all potential XML tree embeddings may be located in different areas of an XML tree. The containment checking process requires that we associate each instance with its absolute location (with respect to the root of the XML tree and its descendants) within the tree. As the SchemaGuide is the metadata equivalent of the XML tree, it provides this absolute location for each instance returned by the embedding process. A SchemaGuide embedding function returns those metadata instances that correspond to the matched instances returned by the XML tree embedding function. We use this one-to-many mapping to reduce the search space of verifying the containment relationship as discussed in §5.4.

In Definition 5.4, we formally define the embedding of a tree pattern in a SchemaGuide. There may be multiple embeddings of a tree pattern in a SchemaGuide, where each embedding defines a mapping between a tree pattern to a subtree of the SchemaGuide with structural characteristics preserved. As will be shown, there is a one-to-many relationship between subtree instances returned from the XML tree embeddings and the subtree instance obtained from a SchemaGuide embedding. Several subtree instances of an XML

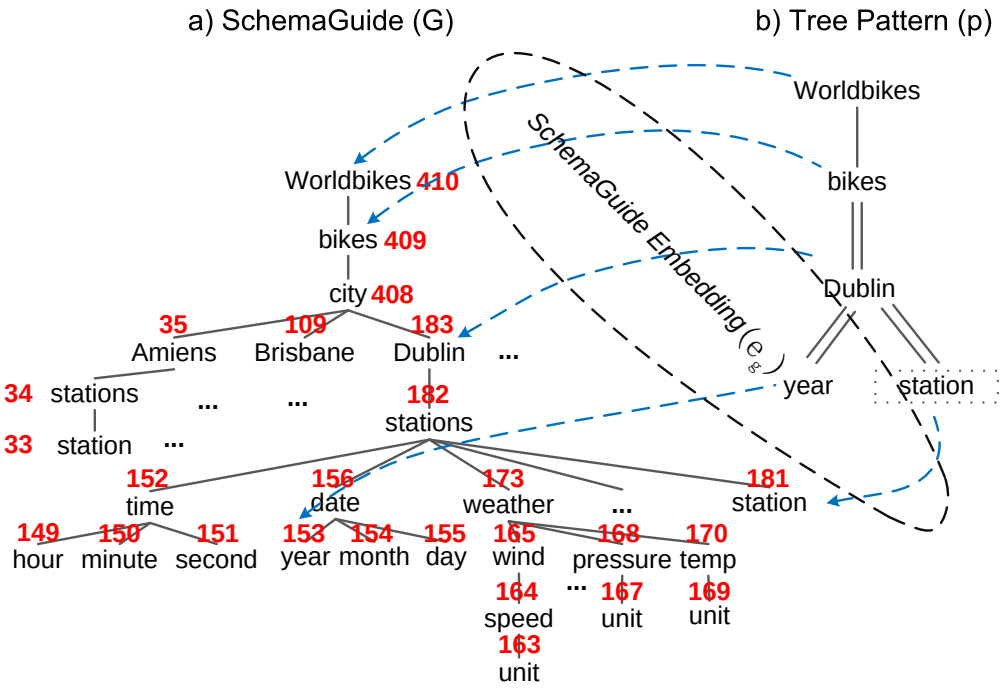


Figure 5.3: SchemaGuide Embedding

tree may map to one single subtree of a SchemaGuide. This optimises the containment checking process by restricting the search space to a set of SchemaGuide subtrees. Definition 5.4 is fundamental to prove the correctness of the containment checking process using a SchemaGuide. Bear in mind that a SchemaGuide G has a tree-based representation with a subtree in G rooted at a schema node u, denoted by g_u.

Definition 5.4 [SchemaGuide Embedding]

Given a tree pattern p and a SchemaGuide G, let $e_g : \mathbb{N}_p \rightarrow \mathbb{N}_G$ be an embedding of p in G where the following characterises are preserved:

- *Root preserving:* $e_g(\mathbb{R}_p) = \mathbb{R}_g$
- *Child edge preserving: if* $(u, v) \in \mathbb{E}_p^{/} \longrightarrow (e_g(u), e_g(v)) \in \mathbb{E}_G$
- *Descendant edge preserving: if* $(u, v) \in \mathbb{E}_p^{//} \longrightarrow (e_g(u), e_g(v)) \in \mathbb{E}_G^{+}$
- *Label preserving:* $\forall u \in \mathbb{N}_p \longrightarrow \mathbb{L}_p(u) = \mathbb{L}_G(e_g(u))$ ■

Example 5.3 (SchemaGuide Embedding e_g in Figure 5.3)

The SchemaGuide embedding e_g maps the root node of p, Worldbikes*, to the root node of g,* Worldbikes *(root preserving). All mapped nodes in p and g have identical labels (label preserving). The relationship between mapped schema nodes in g and the corresponding nodes in p are consistent (child and descendant edge preserving).* ■

The embedding functions serve as a matching of a tree pattern, both at data level (Definition 5.3) and at the schema level (Definition 5.4). As stated previously, containment is based on the evaluation of two tree patterns and the determination of whether one is contained in the other. Definition 5.5 formally defines the containment relationship between two tree patterns. It states that one tree pattern is contained in a second tree pattern if the first result set is a subset of the second result set. As each fragment maps to a tree pattern, we will show in §5.3 that if the first tree pattern is contained in the second tree pattern, then we have shown that the first *fragment* is contained in the second *fragment*. A formal definition of containment between two tree patterns is given in Definition 5.5.

Definition 5.5 [Containment Between Two Patterns]

Given a tree t, two tree patterns p and p' and a SchemaGuide G, where $t \in T_\Sigma$, we say that p is G-contained *in p', denoted by $p \subseteq_G p'$, if for any $t \in T_\Sigma$, $G \vDash t$ and $p(t) \subseteq p'(t)$.* ■

In Definition 5.5, a tree pattern p is said to be *G-contained* (G is a SchemaGuide) in another tree pattern p' if their result sets are contained. Two tree patterns are *equivalent* if $p \subseteq_G p'$ and $p' \subseteq_G p$. From now on, we use the terms *contained* and *G-contained* interchangeably. At this point, we are still focused on structural containment.

For each SchemaGuide embedding, there exists a tree that preserves all characteristics of the embedding. We refer to such a tree as an *Embedded Tree* (see Definition 5.6) derived from the SchemaGuide embedding of a tree pattern in a SchemaGuide. As multiple embeddings may exist where a tree pattern occurs in different parts of the SchemaGuide, there are multiple embedded trees. The concept of the embedded tree allows us to perform containment checking at schema level and thus, reduces the search space. A set of all trees derived from all possible embeddings of p in G is called a *Embedded Tree Set*, denoted by $T_{e_g}(p)$.

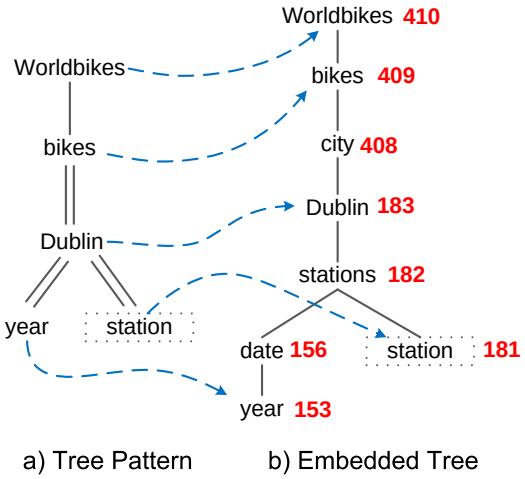


Figure 5.4: Tree Pattern and Embedded Tree

Definition 5.6 [Embedded Tree]

An embedded tree is a tree derived from a SchemaGuide embedding e_g of a tree pattern p in a SchemaGuide G denoted by t_{e_g}. t_{e_g} is obtained as follows:

- *For each node $u \in \mathbb{N}_p$, t_{e_g} contains a node $e_g(u)$ whose label is identical to u. If u is a return node of the tree pattern p, we say that $e_g(u)$ is a* ***return node*** *of t_{e_g}.*
- *Let u be a node in $\mathbb{N}_p$ and v_1, v_2 ... v_k be its children. The node, $e_g(u)$, in t_{e_g} corresponding to u has exactly k children, and for every $1 \leq i \leq k$, its ith child consists of a parent-child chain of nodes, whose labels are those connecting $e_g(u)$ to $e_g(v_i)$.* ■

Definition 5.6 states that an embedded tree t_{e_g} of a tree pattern is obtained from a SchemaGuide through the SchemaGuide embedding such that i) for each node in p, there is a corresponding node in t_{e_g} with an identical label, and the nodes in t_{e_g} that map to the return nodes of p are the return nodes of t_{e_g}; ii) each path in t_{e_g} maps to a path in p, where the root and leaf of the path in p correspond to the root and leaf of the path in t_{e_g}, and the nodes in the path of t_{e_g} contain only parent-child relationship.

An example of the embedded tree is given in Figure 5.4b, where each node in Figure 5.4a maps to a node in Figure 5.4b. The descendant edge is expanded into a chain of parent-child edges, i.e., *Dublin//year* is expanded to *Dublin/stations/date/year* in Figure 5.4b. Furthermore, the *sid* associated with each node in the embedded tree is used to identify the correct

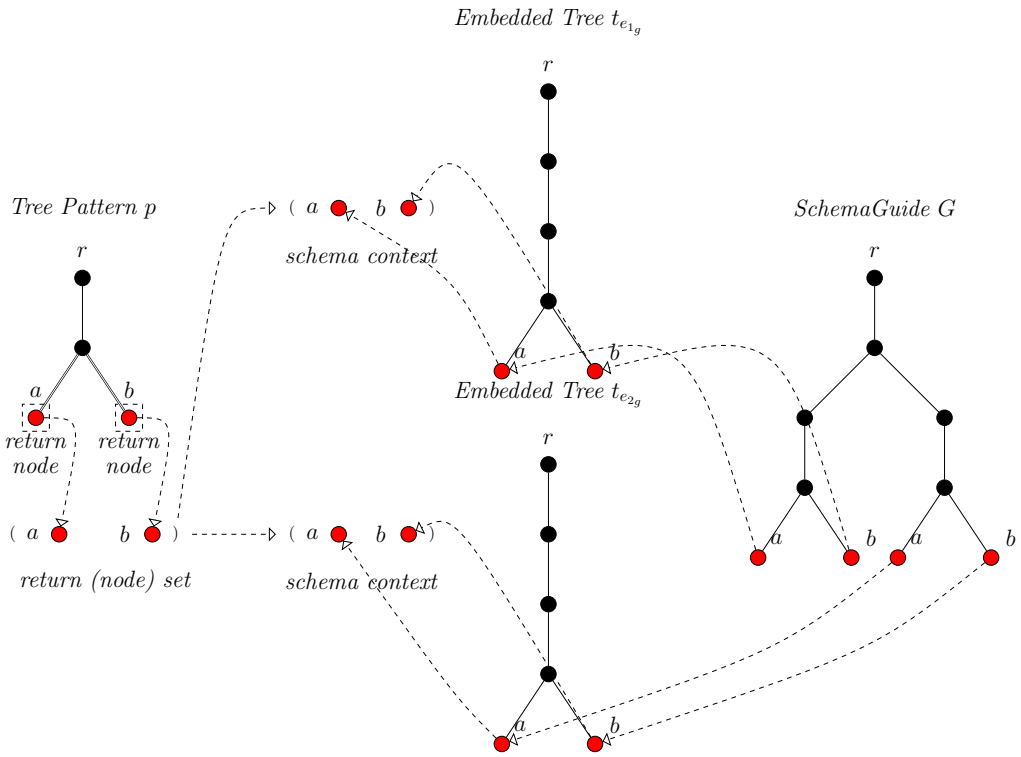


Figure 5.5: Tree Pattern, Embedded Tree and SchemaGuide

instance of nodes with the same label. For instance, *station* may occur multiple times in a SchemaGuide with different sid. In Figure 5.3, *station* within the subtree of *Dublin* and *station* within the subtree of *Amiens* have different sid as the paths between them to the root node of the tree are different according to Definition 5.2. To be precise, in Figure 5.3, *station* has sid values of 33 and 181.

Before proceeding, we now introduce terms that are used extensively in §5.4. As we have discussed, the return set of a tree pattern maps to one or more embedded trees. As shown in Figure 5.5, the return set of p maps to the return nodes of $t_{e_{1_g}}$ and $t_{e_{2_g}}$. We refer to each matched instance of the return set as the *schema context* which contains return nodes of each embedded tree. In other words, different subtrees within the SchemaGuide are often returned, not just a single context.

5.4 Containment Checking Algorithms

As stated in Chapter 2, existing efforts attempt to either reduce the search space or improve the completeness of containment checking in the presence of a DTD, XML Schema or structural summary. However, none of these approaches reflect the actual root-to-node or subtree structure of each instance node within an XML tree, as they provide only "rough" structural information regarding each XML tree. For example, the depth of recursion is unknown where elements are nested inside each other. Furthermore, the subtree structure is unpredictable where an element as defined in the DTD for example, may be optional. As our SchemaGuide provides more detailed information about an XML tree, we will now show that our approach provides more completeness for containment checking and only a small search space is required.

As stated in Definition 5.5, the containment relationship between two tree patterns is verified by determining the relationship between the result sets of them, which tends to be time consuming. To reduce costs, our algorithm uses the embedded tree sets of the two tree patterns rather than the *actual instance nodes* of the return set. To prove the correctness of this method, it is proposed in Proposition 1 that for every result sequence of a tree pattern, there always exists a mapping between the instance nodes within the result sequence and the subtrees in the embedded tree sets. Therefore, the algorithm needs only determine the containment relationship between two tree patterns at the *schema level* based on the mapped subtrees. This is used to prove the correctness of the later propositions that are a fundamental part of our containment checking algorithm.

Proposition 1 states that for each result sequence there exists an embedded tree such that 1) the embedded tree is also a subtree of the corresponding XML tree; and 2) nodes in each schema context of the embedded tree have the same root-to-leaf paths as the result nodes.

Proposition 1 *Let t be a tree and G be a SchemaGuide such that $G \vDash t$, φ is a mapping from t to G and also p be a tree pattern. For each result sequence (u_1, u_2, ..., u_k) $\in p(t)$, there exists a tree t_{e_g} that is derived from an embedding e_g, where $t_{e_g} \in T_{e_g}(p)$, such that:*

- *the embedded tree t_{e_g} is isomorphic to a subtree in t and thus, t_{e_g} is a subtree of t.*
- *the result node u_i ($0 < i \leq k$) is on the path of a node v_i in t_{e_g}, where $v_i \in \mathbb{N}_{t_{e_g}}$ and*

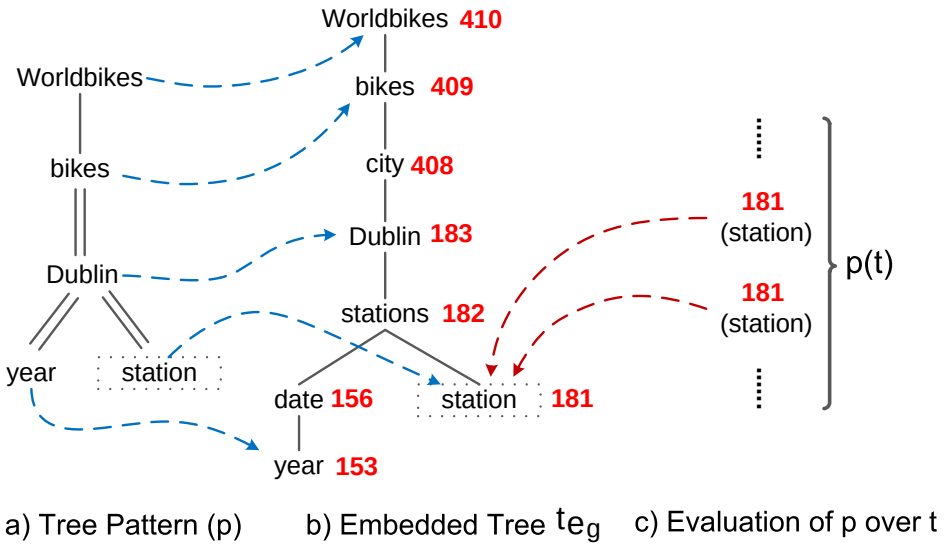


Figure 5.6: Overall Mapping

v_i is the ith return node of t_{e_g}. □

Since the SchemaGuide is derived from an XML tree, based on Property 5.1, we know that there is a mapping between every instance node in the XML tree to schema nodes within the corresponding SchemaGuide. As the tree pattern is a parameter in the function that creates the set of embedded trees in the SchemaGuide, all nodes in the tree pattern are mapped to their equivalent schema nodes in the embedded trees in the SchemaGuide. In many cases, this is a one-to-many mapping as there are likely to be many matched instances of the tree pattern in the SchemaGuide as show in Figure 4.1. Thus, Proposition 1 states that there is a direct mapping between every result node of the tree pattern and the corresponding node in the embedded trees.

PROOF (OF PROPOSITION 1.) [⇒] For each result sequence $(u_1, u_2, \ldots, u_k) \in p(t)$, by definition, there exists an XML tree embedding e of p in t. For every node $m_i \in \mathbb{N}_p$, we have $e(m_i) = u_i$, $0 < i \leq k$. Let e_g be a SchemaGuide embedding of p in G, t_{e_g} is a tree derived from e_g and $t_{e_g} \in T_{e_g}(p)$. For every $v_i \in \mathbb{N}_{t_{e_g}}$ and $v_i = \varphi(u_i)$, where φ is a mapping from t to G, then u_i is on the path of v_i for every $0 < i \leq k$. Therefore, by definition, t_{e_g} is a subtree of the XML tree t. Since $v_i = e(m_i)$ and m_i is the ith return node of p, therefore, v_i is the ith return node of t_{e_g}. ■

The above proof shows that for any result sequence, there exists an embedded tree (subtree)

in the SchemaGuide that is isomorphic to a subtree of the XML tree and each instance node within the result sequence is mapped to a node in the embedded tree with same sid (same root-to-node and subtree structure).

For example, as shown in Figure 5.6, evaluating the tree pattern p (Figure 5.6a) over the XML tree t (Figure 5.1a) results in a sequence of 1-tuple (p has only one return node), where each 1-tuple contains an instance node *station* (Figure 5.6c). For each *station*, there must exist an embedded tree whose return node is also node *station*. In this case, the above proof shows that for each of those 1-tuples, there always exists an embedded tree t_{e_g} derived from the SchemaGuide embedding of p in G, where each 1-tuple maps to the return node of t_{e_g}. In Figure 5.6c, both *station* nodes map to the *station* node of the embedded tree in Figure 5.6b.

PROOF (OF PROPOSITION 1.) **[⇐]** Let (v_1, v_2, . . ., v_k) be the return nodes of p, then by definition, they are also the return nodes of t_{e_g} (with identical labels), where $t_{e_g} \in T_{e_g}(p)$. Let e be an XML tree embedding of p in t, then we have ($e(v_1)$, $e(v_2)$, . . ., $e(v_k)$) $\in p(t)$ and for every $0 < i \leq k$, we have $e(v_i)$ is on the path of v_i. By replacing $e(v_1)$, $e(v_2)$, . . ., $e(v_k)$ with u_1, u_2, . . ., u_k, we have (u_1, u_2, . . ., u_k) $\in p(t)$. ■

The second proof shows the bidirectional nature of the mapping between instance nodes and embedded trees. Thus, for any embedded tree, there is an one-to-many mapping from the embedded tree to the result nodes, e.g., the *station* of the embedded tree in Figure 5.6b is mapped by the two instance nodes resulting from the evaluation of p.

5.4.1 Basic Containment Checking

In this section, we present a basic containment checking algorithm, which iteratively compares two set of embedded trees to verify the containment relationship between two tree patterns, based on Proposition 2. We now show that the basic containment checking needs only to compare the embedded trees retrieved from all possible embeddings of both tree patterns as outlined in Proposition 2.

Proposition 2 *Given a tree t, a SchemaGuide G and two tree patterns p and p', where $t \in T_\Sigma$ and $G \vDash t$, then statements (2-1) and (2-2) are equivalent:*

(2-1) : $p \subseteq_G p'$.

(2-2) : $\forall\, t_{e_g} \in T_{e_g}(p), \exists\, t_{e'_g} \in T_{e'_g}(p') \rightarrow$ *i)* $t_{e'_g}$ *is a subtree of* t_{e_g} *and, ii) both* t_{e_g} *and* $t_{e'_g}$ *have the same return nodes.* □

In (2-1), we state that p is *G-contained* in p'. This is identical to stating that in (2-2), for each embedded tree t_{e_g} in the embedded tree set of p, there exists a tree $t_{e'_g}$ in the embedded tree set of p', such that i) t_{e_g} is a subtree of $t_{e'_g}$ and ii) both of them have the same return nodes (labels are same). Therefore, we can prove that a tree pattern p is contained by another tree pattern p', if each embedded tree of p' is a subtree of an embedded tree of p and both of them have identical return nodes (same labels). We now show that (2-2) can be deduced from (2-1) and vice versa.

PROOF (OF PROPOSITION 2) [(2-1) $\Rightarrow$ (2-2)] Since $p \subseteq_G p'$, so we have $p(t) \subseteq p'(t)$. Let $(u_1, u_2, \ldots, u_k) \in p(t)$, by definition, as $p(t)$ contains a subset of return nodes of $p'(t)$ then we have $(u_1, u_2, \ldots, u_k) \in p'(t)$. Based on Proposition 1, we know that for each $(u_1, u_2, \ldots, u_k) \in p(t)$, there exists a tree t_{e_g}, where $t_{e_g} \in T_{e_g}(p)$. There also exists a tree $t_{e'_g}$, where $t_{e'_g} \in T_{e'_g}(p')$. As $p(t) \subseteq p'$(t), then p and $p^{'}$ have the same return nodes. Therefore, for each $t_{e_g} \in T_{e_g}(p)$, t_{e_g} must contain a subtree $t_{e'_g}$ in $T_{e'_g}(p')$ with the same return nodes so that $p(t) \subseteq p'$(t) is valid. ■

This proof outlines the fact that if p is contained within p', for any embedded tree t_{e_g} of p, there must exist an embedded tree t_{e_g} in p so that $t_{e'_g}$ is a subtree of t_{e_g} that have the same return nodes.

PROOF (OF PROPOSITION 2) [(2-2) $\Rightarrow$ (2-1)] Let t be a tree, and the result sequence $(u_1, u_2, \ldots, u_k) \in p(t)$. There exists a tree $t_{e_g} \in T_{e_g}(p)$, t_{e_g} is a subtree of t. Since t_{e_g} contains a subtree $t_{e'_g} \in T_{e'_g}(p')$ with the same return nodes and $t_{e'_g}$ is a subtree of t, by Proposition 1, $(u_1, u_2, \ldots, u_k) \in p'(t)$. Therefore, $p(t) \subseteq p'(t)$, which further implies $p \subseteq p'$ based on Definition 5.5. ■

The above proof states that for any embedded tree t_{e_g} of p, there exists an embedded tree $t_{e'_g}$ in p' so that $t_{e'_g}$ is a subtree of t_{e_g} that have the same return nodes, then the result of p over t is a subset of the result of p' over t, which indicates that p is contained in p' (Definition 5.5).

According to Proposition 2, the containment relationship between tree patterns p and p' is determined by the following two steps:

1. Generating the embedded tree sets of p and p', $T_{e_g}(p)$ and $T_{e'_g}(p')$, according to the SchemaGuide embeddings.

2. Comparing all trees in the embedded tree sets and then verifying whether (2-2) of Proposition 2 is true or not.

The complexity of the algorithm is $O(|\mathbb{N}_G|^2 \times |T_{e_g}(p)| \times |T_{e'_g}(p')|)$. For p and p', there are maximum $|\mathbb{N}_G| \times |T_{e_g}(p)|$ and $|\mathbb{N}_G| \times |T_{e'_g}(p')|$ nodes, respectively. Therefore, to compare embedded trees derived from all embeddings of both tree patterns, the time required is $|\mathbb{N}_G|^2 \times |T_{e_g}(p)| \times |T_{e'_g}(p')|$.

5.4.2 Optimised Containment Checking

Proposition 2 highlights the fact that the algorithm requires two iterations for checking containment: one for each embedded tree set. To further improve performance, Proposition 3 implies a new algorithm, which reduces the time complexity by evaluating the first tree pattern over the embedded trees retrieved from the second tree pattern. This algorithm can be achieved in a more efficient way by using holistic twig pattern matching algorithms such as, TwigList [QYD07] and OTwig [LR10], which cannot be achieved using the former approach.

Proposition 3 *Given a tree t, a SchemaGuide G and two tree patterns p and p', where $t \in T_\Sigma$ and $G \vDash t$, then (3-1) and (3-2) are equivalent,*

(3-1) : $\forall t_{e_g} \in T_{e_g}(p)$ *whose return nodes are* $(u_1, u_2, \ldots, u_k)$*, there exists* $(u_1, u_2, \ldots, u_k) \in p'(t_{e_g})$*.*

(3-2) : $\forall\, t_{e_g} \in T_{e_g}(p), \exists\, t_{e'_g} \in T_{e'_g}(p') \rightarrow$ *i)* $t_{e'_g}$ *is a subtree of* t_{e_g} *and, ii) both* t_{e_g} *and* $t_{e'_g}$ *have the same return nodes.* □

Proposition 3 states that for each embedded tree t_{e_g} of p, if the return nodes of t_{e_g} are identical to the result nodes p' over t_{e_g}, then p is contained in p'. To show the correctness of this proposition, we prove that (3-1) is equivalent to (3-2). To demonstrate the equivalence,

we first show that (3-2) can be deduced from (3-1) of Proposition 3. Note that, (3-2) is equivalent to (2-2) of Proposition 2.

PROOF (OF PROPOSITION 3.) [(3-1) $\Rightarrow$ (3-2)] Since $(u_1, u_2, \ldots, u_k) \in p'(t_{e_g})$, based on Proposition 1, there must exist a tree which is a subtree of t_{e_g} and let such a tree be $t_{e'_g}$. As u_i is on the path of v_i in $t_{e'_g}$, where $v_i \in \mathbb{N}_{t_{e'_g}}$, $0 < i \leq k$, thus, t_{e_g} and $t_{e'_g}$ have the same return node. ■

The above proof verifies that if the nodes returned by evaluating a tree pattern p' over the embedded tree t_{e_g} are equivalent to the return nodes of t_{e_g} of p, then (3-2) holds. In the following proof, we also show that the reverse holds.

PROOF (OF PROPOSITION 3.) [(3-2) $\Rightarrow$ (3-1)] Let $(u_1, u_2, \ldots, u_k)$ be the return nodes of p. By Definition 5.6, $(u_1, u_2, \ldots, u_k)$ are the return nodes of all trees derived from the SchemaGuide embeddings of p in G. Since $\forall t_{e_g} \in T_{e_g}(p)$, there exists a tree $t_{e'_g}$ that is derived from a SchemaGuide embedding of p' so that $t_{e'_g}$ is a subtree of t_{e_g} and t_{e_g} and $t_{e'_g}$ have the same return nodes. Therefore, $(u_1, u_2, \ldots, u_k) \in p'(t_{e_g})$. ■

The above proof states that for an embedded tree t_{e_g} of p, there exists an embedded tree $t_{e'_g}$ of p', where $t_{e'_g}$ is a subtree of t_{e_g} and both subtrees have the same return nodes. Thus, (3-1) in Proposition 3 is true. We can exploit this property to deliver an optimised containment checking algorithm.

Based on Proposition 3, the following algorithm checks the containment relationship between p and p':

- Retrieving all embedded trees of p that are derived from the embeddings of p in a SchemaGuide G.
- Evaluating p' over all $t_{e_g} \in T_{e_g}(p)$ and verifying that the return nodes of p belong to $p'(t_{e_g})$.

The complexity of the algorithm is $O(|\mathbb{N}_G| \times |T_{e_g}(p)| \times |\mathbb{N}_{p'}|)$ as each embedded tree of p contains at most $|\mathbb{N}_G|$ nodes and the evaluation of p' takes $|\mathbb{N}_{t_{e_g}}| \times |\mathbb{N}_{p'}|$, which equals to the number of nodes in the embedded tree multiplies the number of nodes in the tree pattern p'.

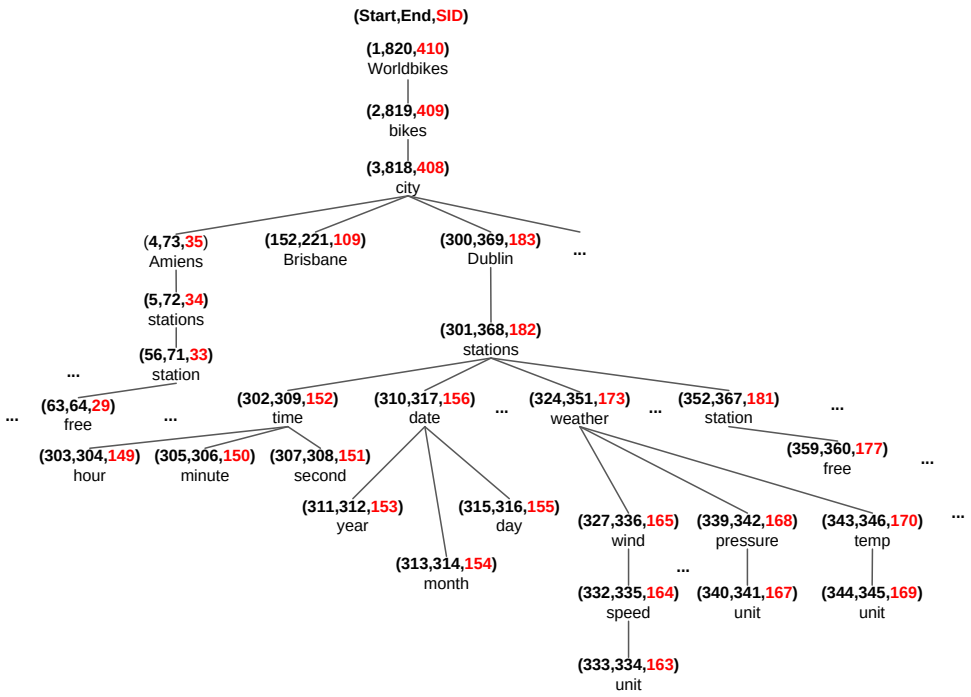
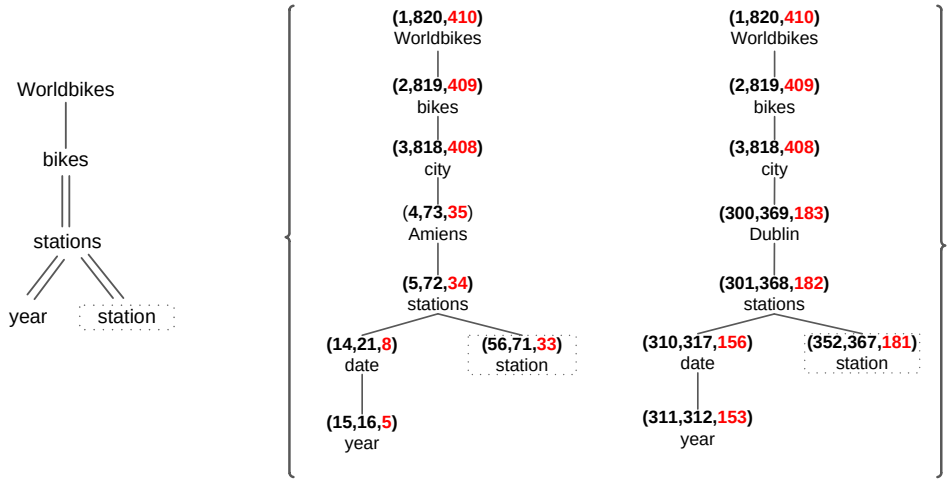


Figure 5.7: A SchemaGuide With Positional Encoding Scheme

5.4.3 Region-Based Optimisation

The algorithms described in the previous section must consider all nodes, either in tree patterns or in all embedded trees. We now present an approach that reduces the search space by focusing only on the return nodes of each embedded tree (schema context) during containment checking. The idea is to encode each schema node by applying a *positional (region) encoding/labelling scheme*, the benefit of this is that we can quickly determine regions within a SchemaGuide in where each schema node locates and then check containment relationship based on those regions.

The region encoding scheme we adopted in our approach is called the *StartEnd* encoding scheme, which was originally devised for XML twig pattern matching algorithms to facilitate the process of determining the parent-child and ancestor-descendant relationships between XML tree nodes and it is a variation of the *PrePost* encoding scheme [Gru02]. Although *StartEnd* is used, it is not mandatory as any containment-based encoding scheme, as discussed in [OR10], can be adopted. As shown in Figure 5.7, each schema node is now represented by a 3-tuple, (*start*,*end*,*sid*), where the *start* and *end* values of a schema node



(a) Tree Pattern p_1 **(b)** Embedded Tree Set $T_{e_{1_g}}(p_1)$

Figure 5.8: Sample 1: Tree Pattern and Embedded Tree Set

u forms the *region* of u, denoted by $\texttt{REG}(u)$. The *start* and *end* values can be obtained by performing a depth-first traversal and sequentially assigning a number to each visit. Each node is visited twice, once before visits all its children and once after. Although leaf nodes are visited only once, we treat them as if they were visited twice so that, rather than having just the start value, they are assigned both start and end values.

Based on the region encoding scheme, schema nodes can be divided into different regions as outlined in Definition 5.7 and explained in Example 5.4.

Definition 5.7 [Region Containment]

Give two schema nodes u *and* v, u *is* ***in the region of*** v *iff* $u.start > v.start$ *and* $u.end < v.end$, *denoted by* $\texttt{REG}(u) \subseteq \texttt{REG}(v)$. ■

Example 5.4 (Region Containment Between *stations*, *Amiens* and *Dublin*)

As shown in Figure 5.7, by definition, node stations*, labelled with (5,72,34), is in the region of node* Amiens *(4,73,35) as region (5,72) is in the region of (4,73). Moreover,* stations *(301,368,182) is in the region of* Dublin *(300,369,183) as (301,368) is in the region of (300,369). In brief, the start value 301 is greater than 300 and the end value 369 is less*

than 369. From an optimisation perspective, it was not necessary to check all nodes within that region as it is replaced with a single comparison operation. ■

Definition 5.8 outlines that the region between two schema nodes are same only if the two schema nodes are identical (with same sid). This forms the basis of the region-based containment checking.

Definition 5.8 [Region Equivalent]
*Given two schema nodes u and v, u and v have **equivalent regions** iff $u.start = v.start$ and $u.end = v.end$, denoted by $\texttt{REG}(u) \equiv \texttt{REG}(v)$.* ■

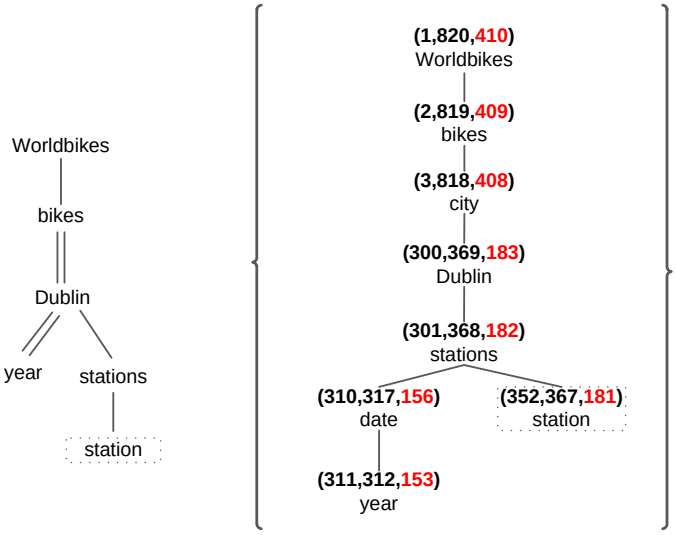
Proposition 4 outlines the fact that given two tree patterns, p and p', to prove that p' contains p ($p \subseteq p'$), it is necessary to verify only whether (4-2) of Proposition 4 is satisfied. If two tree patterns are contained, the corresponding set of schema contexts are contained and (4-2) states that their nodes are equivalent.

Proposition 4 *Given a tree t, a SchemaGuide G and two tree patterns p and p', where $t \in T_\Sigma$ and $G \vDash t$, let $t_{e_g} \in T_{e_g}(p)$, $t_{e'_g} \in T_{e'_g}(p')$ and the return nodes of t_{e_g} and $t_{e'_g}$ are (u_1, u_2, …, u_k) and (v_1, v_2, …, v_k), respectively, the following statements are equivalent,*
(4-1) : $p \subseteq_G p'$
(4-2) : $\forall t_{e_g} \in T_{e_g}(p), \exists t_{e'_g} \in T_{e'_g}(p') \rightarrow \texttt{REG}(u_i) \equiv \texttt{REG}(v_i),\ 0 < i \leq k$ □

(4-2) of Proposition 4 shows that p' contains p if the schema contexts of p are a subset of the schema contexts of p' (see Example 5.5).

Example 5.5 (Subtree containment between p_1 and p_2 in Figure 5.8 and Figure 5.9)
It is easy to determine that p_1 contains p_2 ($p_2 \subseteq p_1$), as the number of return nodes of the embedded tree set ($T_{e_{2_g}}(p_2)$) of p_2 is a subset of the return nodes of the embedded tree set ($T_{e_{1_g}}(p_1)$) of p_1. The result is that $\{$station (352,367,181)$\} \subseteq \{$station (56,71,33), station (352,367,181)$\}$. ■

We now demonstrate that if two tree patterns are contained, then the return nodes of the embedded trees of the first tree pattern are a subset of the return nodes of the embedded trees of the second tree pattern.



(a) Tree Pattern p_2 **(b)** Embedded Tree Set $T_{e2_g}(p_2)$

Figure 5.9: Sample 2: Tree Pattern and Embedded Tree Set

PROOF (OF PROPOSITION 4.) [(4-2) $\Rightarrow$ (4-1)] For all $u_i \in (u_1, u_2, \ldots, u_k) \in t_{e_g}$, there exists a node $v_i \in (v_1, v_2, \ldots, v_k) \in t_{e'_g}$ such that $\texttt{REG}(u_i) \equiv \texttt{REG}(v_i)$. According to Definition 5.8, since $\texttt{REG}(u_i) \equiv \texttt{REG}(v_i)$, u_i is equivalent to v_i, and $(u_1, u_2, \ldots, u_k)$ is equivalent to $(v_1, v_2, \ldots, v_k)$. This implies that p and p' have same return nodes. According to the definition of the SchemaGuide embedding, u_i and v_i have identical root-to-node and subtree structure. Therefore, the embedded trees corresponding to u_i and v_i have identical structure. As a result, according to Proposition 2, $p \subseteq_G p'$. ■

PROOF (OF PROPOSITION 4.) [(4-1) $\Rightarrow$ (4-2)] Let $t_{e_g} \in T_{e_g}(p)$ and $t_{e'_g} \in T_{e'_g}(p')$. Since $p \subseteq_G p'$ and according to Proposition 2, for all t_{e_g} in $T_{e_g}(p)$, there exists a $t_{e'_g}$ in $T_{e'_g}(p')$, such that $t_{e'_g}$ is a subtree of t_{e_g} and both subtrees have the same return nodes. Therefore, those return nodes are within the same regions and for all $u_i \in (u_1, u_2, \ldots, u_k) \in t_{e_g}$, there exists a $v_i \in (v_1, v_2, \ldots, v_k) \in t_{e'_g}$, where $0 < i \leq k$, we have $\texttt{REG}(u_i) \equiv \texttt{REG}(v_i)$. ■

We now present an algorithm based on Proposition 4 to check the containment relationship between two tree patterns p and p'. The algorithm is comprised of two steps.

1. Retrieving all embedded trees $T_{e_g}(p)$ and $T_{e'_g}(p')$ of p and p' that are derived from the SchemaGuide embedding of p and p', respectively.

2. Check whether or not (4-2) of Proposition 4 is satisfied.

The above algorithm requires two nested iterations, one traverses all return nodes of $t_{e_g} \in T_{e_g}(p)$ and the other one loops through all return nodes of $t_{e'_g} \in T_{e'_g}(p')$. The algorithm returns either true or false indicating whether p is contained in p' or not. We use a function RET_NUM which takes a tree pattern as input and returns the number of return nodes of a tree pattern. RET_NUM(p) and RET_NUM(p') output the number of return nodes of p and p', respectively. The complexity of the algorithm is as listed below,

$$O(|T_{e_g}(p)| \times \texttt{RET\_NUM}(p) \times |T_{e'_g}(p')| \times \texttt{RET\_NUM}(p'))$$

which is more efficient than the previous approach which requires $O(|\mathbb{N}_G| \times |T_{e_g}(p)| \times |\mathbb{N}_{p'}|)$, where RET_NUM($p'$) $\leq \mathbb{N}_{p'}$ as the number of return nodes of p' is always less equal than the total number of nodes in $\mathbb{N}_{p'}$ and RET_NUM(p)$\times|T_{e'_g}(p')| \leq |\mathbb{N}_G|$. In the worst case, the number of return nodes of p and p' are equal to the number of nodes in G and the number of embedded trees is one. In reality, it is unusual that a query returns every node in an XML tree.

5.4.4 Subtree-Based Containment

Our region optimised algorithm has clear benefits to containment checking in terms of search space reduction, as will be demonstrated later in our evaluation. However, according to the W3C recommendation [W3C10d], the result of an XML query is a new XML document (XML tree) constructed by concatenating subtrees of all result nodes. Therefore, we consider that if the result nodes of one tree pattern are within the subtree of the result nodes of the second one, then they are contained. This is an essential concept to our fragment-based approach as it determines the containment relationship between two fragments, whether the first fragment should reside at the level above or below the second one. This is different from standard containment checking which checks whether one set of result sequences is contained in another. To differ from the previous terminology, we refer to this type of containment relationship as *subtree containment*. Given two tree patterns p and p', we say that p is subtree-contained in p' (see Definition 5.9), denote by $p \subseteq_G^{Sub} p'$. As shown later, to verify the containment relationship between two fragments, it can use either

concept to determine containment.

Definition 5.9 states that two tree patterns are *subtree-contained* if all result nodes of the first tree pattern are within the region of the result nodes of the second tree pattern.

Definition 5.9 [Subtree Containment Between Tree Patterns]

Given two tree patterns p and p', an XML tree t and a SchemaGuide G, where $G \vDash t$. $p \subseteq_G^{Sub} p'$ if for each $u_i \in (u_1, u_2, \ldots, u_k) \in p(t)$ there exists $v_i \in (v_1, v_2, \ldots, v_k) \in p'(t)$ such that u_i is in the subtree of v_i, where $0 < i \leq k$. ■

Based on Definition 5.9, Proposition 5 implies a containment checking algorithm which determines whether the return nodes of the first embedded tree set are within at least one return node of the second embedded tree set.

Proposition 5 *Given a tree t, a SchemaGuide G and two tree patterns p and p', where $t \in T_\Sigma$ and $G \vDash t$, let $t_{e_g} \in T_{e_g}(p)$, $t_{e'_g} \in T_{e'_g}(p')$ and the return nodes of t_{e_g} and $t_{e'_g}$ are (u_1, u_2, …, u_k) and (v_1, v_2, …, v_k), respectively, (5-1) and (5-2) are equivalent,*

(5-1) : $p \subseteq_G^{Sub} p'$

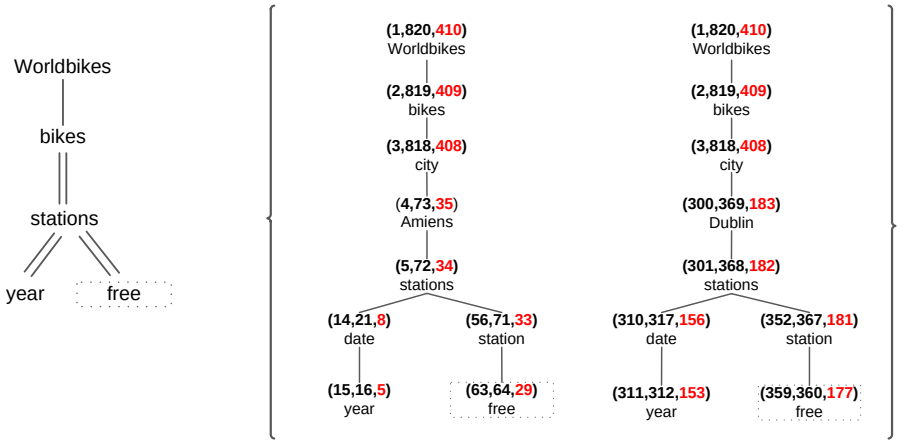
(5-2) : $\forall t_{e_g} \in T_{e_g}(p), \exists t_{e'_g} \in T_{e'_g}(p') \rightarrow \mathtt{REG}(u_i) \subseteq \mathtt{REG}(v_i)$, where $0 < i \leq k$. □

(5-2) of Proposition 5 indicates that p is subtree-contained in p' if for each embedded tree t_{e_g} of p, there exists an embedded tree $t_{e'_g}$ of p' such that the return nodes of t_{e_g} are within the subtree of the return nodes of $t_{e'_g}$.

Example 5.6 (Subtree-Containment Between p_1 and p_3 in Figure 5.8 and 5.10)

The return node of p_1 and p_3 are station *and* free*, respectively. In this case, free (63,64,29) and free (359,360,177) (Figure 5.10b) are within the region of station (56,71,33) and station (352,367,181) (Figure 5.8b), respectively.* ■

We now provide our proofs to demonstrate if two tree patterns are contained, then the return nodes of all embedded trees of the first tree pattern is in the region of the return nodes of the embedded tree of the second tree pattern.



(a) Tree Pattern p_3 **(b)** Embedded Tree Set $T_{e_{3_g}}(p_3)$

Figure 5.10: Sample 3: Tree Pattern and Embedded Tree Set

PROOF (OF PROPOSITION 5) [(5-2) $\Rightarrow$ (5-1)] Suppose $(n_1,n_2,\cdots,n_k) \in p(t)$, $(n'_1,n'_2,\cdots,n'_k)$ $\in p'(t)$ are the result nodes of p and p', respectively. Let φ be a mapping function of t in a SchemaGuide G, where $G \vDash t$. By definition, we have $(\varphi(n_1),\varphi(n_2),\cdots,\varphi(n_k))$ $\in t_{e_g}$ and $(\varphi(n'_1),\varphi(n'_2),\cdots,\varphi(n'_k)) \in t_{e'_g}$ and we replace them with $(u_1, u_2, \ldots, u_k)$ and $(v_1, v_2, \ldots, v_k)$, respectively. Since for all $u_i \in (u_1, u_2, \ldots, u_k) \in t_{e_g}$, there exists a node $v_i \in (v_1, v_2, \ldots, v_k) \in t_{e'_g}$ such that $\texttt{REG}(u_i) \subseteq \texttt{REG}(v_i)$ and u_i is within the subtree of v_i. As for each $u_i \in (u_1, u_2, \ldots, u_k)$, there exists a $v_i \in (v_1, v_2, \ldots, v_k)$ such that u_i is in the subtree of v_i. By definition, n_i and n'_i are on the same path of u_i and v_i, respectively. As a result, for all $n_i \in (n_1,n_2,\cdots,n_k)$ and $n'_i \in (n'_1,n'_2,\cdots,n'_k)$, n_i is in the subtree of n'_i. Therefore, according to Definition 5.9, $p \subseteq^{Sub}_{G} p'$. ■

PROOF (OF PROPOSITION 5) [(5-1) $\Rightarrow$ (5-2)] Let $(u_1, u_2, \ldots, u_k) \in p(t)$, $(v_1, v_2, \ldots, v_h)$ $\in p'(t)$ and φ be a mapping function of t to a SchemaGuide G. For all $t_{e_g} \in T_{e_g}(p)$ and $t_{e'_g} \in T_{e'_g}(p')$, we have $(\varphi(u_1),\varphi(u_2),\cdots,\varphi(u_k)) \in t_{e_g}$ and $(\varphi(v_1),\varphi(v_2),\cdots,\varphi(v_k))$ $\in t_{e'_g}$. By definition, for all $u_i \in (u_1, u_2, \ldots, u_k) \in p(t)$, there exists a node v_i such that $v_i \in (v_1, v_2, \ldots, v_k) \in p'(t)$ and u_i is in the subtree of v_i $(0 < i \leq k)$. Since u_i, v_i are on the path of $\varphi(u_i)$ and $\varphi(v_i)$, respectively, therefore, $\varphi(u_i)$ is in the subtree of $\varphi(v_i)$, that is, $\texttt{REG}(\varphi(u_i)) \subseteq \texttt{REG}(\varphi(v_i))$. ■

Based on Proposition 5, we can devise an approach that uses two steps to check whether a tree pattern p is *subtree-contained* in a second tree pattern p':

- Retrieving all embedded trees $T_{e_g}(p)$ and $T_{e'_g}(p')$ of p and p' that are derived from the SchemaGuide embedding of p and p', respectively.
- Check whether or not (5-2) of Proposition 5 is satisfied.

The time complexity required for the containment checking is as follows,

$$O(|T_{e_g}(p)| \times \texttt{RET\_NUM}(p) \times |T_{e'_g}(p')| \times \texttt{RET\_NUM}(p'))$$

5.4.5 Incorporating Value Predicates

The containment algorithm presented so far does not consider that predicates may appear in a fragment/tree pattern. A predicate, denoted by *pred*, is represented in the form of val θ c, where val is the value of the node, $\theta \in \{<, =, >, \leq, \geq, \neq\}$, and c is a constant value. Given a tree pattern p mapping to a fragment f, we define a binding function ϕ, which associates the predicates with the corresponding nodes in p. The logical formula of the binding is represented by ϕ_u(*pred*), where $u \in \mathbb{N}_p$ and *pred* is a predicate in p. If a node has no predicate, the binding function assigns an empty value to it. For each node u in p, its corresponding schema node within each embedded tree of p "inherits" its value predicate during the SchemaGuide embedding. The embedding of the node *year* in Figure 5.11a is mapped to the node *year* in Figure 5.11b and its value predicate is "inherited". The remaining nodes in Figure 5.11a do not have a predicate and therefore, are assigned empty values as shown in Figure 5.11b.

Given two tree patterns p and p', where $\mathcal{P}$ and $\mathcal{P}'$ are sets of value predicates (including empty predicates) of p and p', respectively, $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2 \cup \ldots \cup \mathcal{P}_n$ and $\mathcal{P}' = \mathcal{P}'_1 \cup \mathcal{P}'_2 \cup \ldots \cup \mathcal{P}'_m$. Containment checking is achieved by comparing all value predicates within $\mathcal{P}$ and $\mathcal{P}'$ in sequential order. During containment checking, value predicates are processed first and only when a TRUE value is returned, is structural checking then performed.

The containment checking algorithm first compares each pair of value predicates corresponding to each pair of mapped schema nodes between embedded trees of two tree patterns. If comparison takes place between $\mathcal{P}_i$ $(0 < i \leq n)$ and $\varnothing$ or $\varnothing$ and $\mathcal{P}'_j$ $(0 < j \leq m)$,

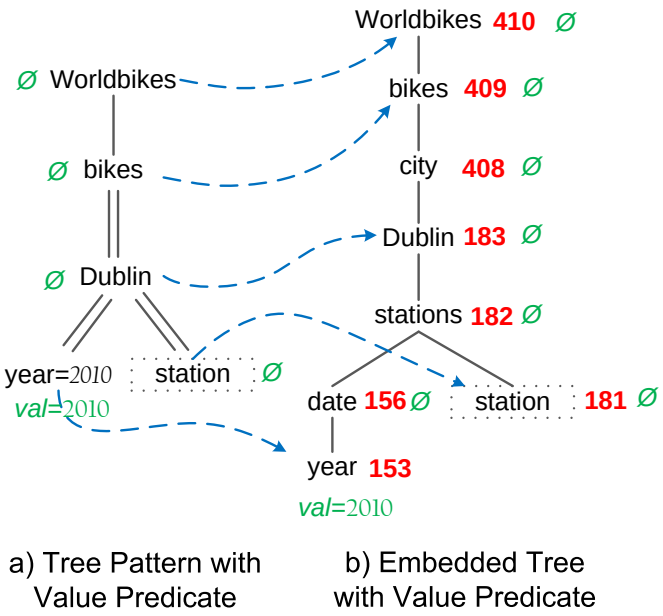


Figure 5.11: Tree Pattern and Embedded Tree with Predicates

then FALSE is returned as one node contains no predicate. If both nodes have associated predicates, a string-based or numeric-based comparison is performed.

5.4.6 Containment at XFM Graph Level

Using previous definitions and propositions, Definition 5.10 states the requirements that are necessary to determine the containment relationship between fragments within an XFM view graph. It is the key component of the graph construction and the view adaptation process that we will introduce in Chapter 7.

Definition 5.10 [Containment Between Fragments]

Given a tree t and a SchemaGuide G, where $G \vDash t$, f and f' are fragments and p and p' are tree patterns mapping to f and f', the fragment f is contained by the fragment f' if either 1 or 2 is true:

1. $p \subseteq_G p'$
2. $p \subseteq_G^{Sub} p'$ ■

Recall that each fragment maps to a tree pattern. The approach is to verify the containment relationship between two fragments by determining the corresponding containment relationship between two tree patterns that map to these fragments.

Containment Rules Between Fragments. Based on the characteristics of the XFM view graph, there are cases where the containment relationship is obvious and thus, it is sufficient to simply follow the set of predefined *containment rules*. Given two fragments f and f', the following rules can be used together with the containment algorithms to check whether f contains f', *true* for f contains f', otherwise, *false*:

- if f is a RF fragment and f' is any fragment type, return *true*.
- if f' is a RF fragment and f is not a RF fragment, return *false*.
- if f' is a VF fragment and f and f' are shared by at least one view, return *true*.
- if f' is a VF fragment and f and f' are shared by different views, then return *false*.
- if f is a VF fragment, return *false*.

5.5 Summary

In this chapter, we presented our containment checking process. Unlike containment checking in relational systems, this is far more complex due to the tree-structured nature of XML documents and XPath queries. The challenge is to determine if one fragment (or step in an XPath expression) is contained within another and then to provide a mechanism whereby the containment check could be optimised. As part of this process, we developed the SchemaGuide and through a number of mapping and embedding functions were able to reduce the search space of the containment check considerably. The goal of this chapter was to demonstrate the correctness of our approach. Later, in Chapter 7, we show how containment checking algorithm is used together with the view adaptation algorithms to manage view redefinitions.

Chapter 6

A Fragment Selection Strategy

In Chapter 4, we introduced the XFM view graph, where views are integrated with fragments shared and each fragment within the XFM view graph represents a candidate for materialisation. Although materialising every fragment for every view will have the best performance for query processing, the view adaptation costs will be far higher. As a result, a fragment selection strategy is necessary to determine a suitable set of fragments for materialisation. In §6.1, an overview of the selection strategy is presented together with a brief description of each component of the system. We then proceed in §6.2 with a description of the full range of costs that are estimated, based on the view graph. These costs are then used by the heuristics introduced in §6.3 to manage the selection of fragments. Our three selection mechanisms are then presented in §6.4. The Graph-Based, Cluster-Based and Subgraph-Based selection strategies ensure that an optimised selection plan is produced and *all* views have at least one fragment materialised.

6.1 Fragment Selection Overview

We begin with a high level overview of the fragment selection process [LR11], where the objective is to select candidate fragments for materialisation. From a high level point of view, Figure 6.1 shows the selection method as consisting of two processes, Fragment Selection and View Coverage.

- **Fragment Selection.** The purpose of Fragment Selection is to perform a baseline

scan over the entire fragment list to select an initial set of fragments for materialisation. The end result is two separate fragment lists: the first contains a set of fragments to materialise; the second contains those fragments which have not been selected, but which are now passed to the View Coverage process where new criteria is used to select additional fragments for materialisation.

- **View Coverage.** As fragments chosen by the Fragment Selection process may not cover all views, the View Coverage process invokes an iterative-based process to ensure that fragments from unrepresented views are part of the final materialisation set. This process will not be invoked if fragments selected in Fragment Selection process represent every view.

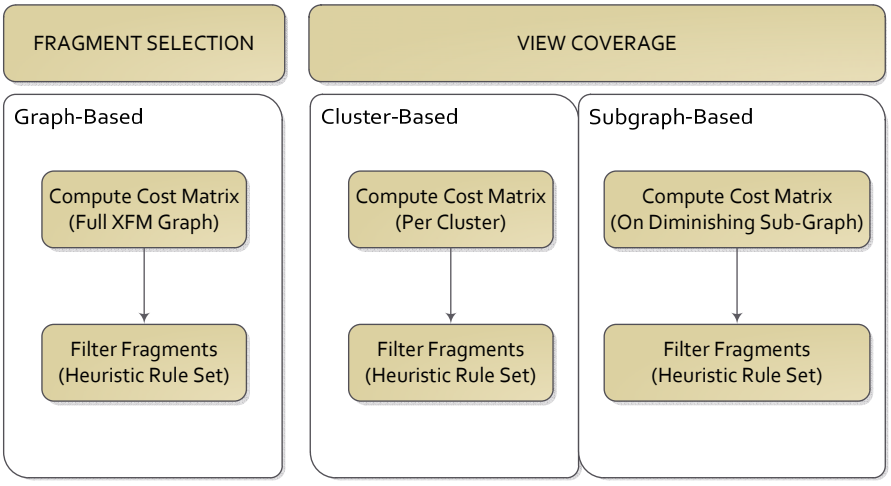


Figure 6.1: Fragment Selection Methodology

Within both processes, there are two components which drive the fragment selection process.

- **Cost Estimation.** The Cost Estimation process computes different forms of costs for each fragment based on a specified context. As shown in Figure 6.1, in Fragment Selection, the costs are computed based on the global view graph, whereas in View Coverage, costs are calculated based on a subset of existing views (Per Cluster and On Diminishing Sub-Graph).

- **Filtering.** The Filtering process applies a set of cost-based heuristic rules to each

fragment and only those fragments that satisfy the rules are added to the candidate list for materialisation.

6.2 Selection Metrics

In selecting fragments, different costs are computed for each fragment and a *cost matrix* is used to map costs to fragments. The cost-based heuristics, to be presented in §6.3, use a combination of different costs from the matrix. Table 6.1 gives an illustration of the cost matrix of fragments in the XFM view graph shown in Figure 4.5 on Page 54. Before we give a detailed description of the cost matrix, we first present a set of primitive costs that form the basis of the cost matrix and the cost-based heuristics.

FID	*Shareability*	*Maintenance Cost*	*Reuse Cost*	*Materialisation Cost*	*Cardinality*	*Fragment Usage*	*Frequency*	*Fragment Cost*	*Global Benefit*
DF1	5	380275484	1	380275485	1	77	0.09	1	1
FF7	5	27222192	39048	27300292	39048	77	0.09	78100	1
FF2	4	27222192	758917404	27378388	39048	75	0.09	156196	0.8
DF4	3	27183321	505957833	1545135277	38870	69	0.08	1517951956	0.6
VF1	1	359660967	0	2858942723303	24100889	3	0.01	2858583062336	0.2

Table 6.1: Cost Matrix

- **Shareability of a fragment** f (**`share`**(f)): the number of views sharing fragment f in the XFM view graph.
- **Maintenance cost of a fragment** f (**`maincost`**(f)): is the storage requirement of materialising f. In our approach, the storage requirement is the total number of nodes to be materialised.
- **Reuse cost of a fragment** f (**`reusecost`**(f)): the cost of reusing a materialised fragment f rather than computing it from scratch for every change. For instance, to determine the cost of the operation node between FF8 and FF9 as shown in Figure 4.5, the process can use materialised data in FF8 rather than accumulating the costs of operation nodes and fragment nodes between RF fragment and FF8.

- **Materialisation cost of a fragment f (`matcost`(f))**: the cost of materialising f.
- **Cardinality of a fragment f (`card(`f`)`)**: the number of instance nodes within f.

Apart from the above costs, we also consider user preferences relative to each view, e.g., whether one view is queried more than another. These statistics are used to further improve a fragment's chances of selection, where the fragment is frequently requested by the user.

- **Usage of a view V (`usage(`V`)`)**: the number of times a view V is used or is required/queried by users.
- **Usage of a fragment f (`usage(`f`)`)**: the number of times a fragment is required. This value is with respect to the number of views sharing f and their corresponding usages. The usage of a fragment f is equal to the sum of the usages of all views sharing f,

$$\texttt{usage(f)} \quad = \quad \sum_{i=0}^{k} \texttt{usage}(V_i), \text{ where } V_i \text{ shares } f. \tag{6.1}$$

- **Frequency of a fragment f (`freq`(f))**: measures the frequency of the usage of f among the overall fragment usages. The fragment frequency is equal to the result of dividing the fragment usage of f by the overall fragment usages,

$$\texttt{freq}(f) \quad = \quad \frac{\texttt{usage}(f)}{\sum_{i=0}^{k} \texttt{usage}(f_i)} \tag{6.2}$$

  where k is equal to the number of fragments in an XFM view graph and $\texttt{freq}(f)$ is in the range $(0, 1]$.

Cost of Operation and Fragment Nodes. Recall that views are formed by operation nodes and fragment nodes. The cost for an entire view is equivalent to the cost of all nodes along the path between the RF fragment and the VF fragment. For example, in Figure 4.5, the cost of evaluating V_4 is equal to the sum of the costs of all nodes along the path from

RF to VF4. We use the polymorphic function `cost`, to calculate operation nodes (o) and fragment nodes (f) in views:

$$\begin{aligned} \texttt{cost}(o) &= \text{the execution cost of } o \text{ + the cost of the fragment node} \quad (6.3) \\ &\quad \text{at } \texttt{Level}(o)\text{-1} \\ \texttt{cost}(f) &= \text{the cost of the operation node at } \texttt{Level}(f)\text{-1} \quad (6.4) \\ \texttt{cost}(f_p, f) &= \text{the sum of the costs of fragment and operation nodes} \quad (6.5) \\ &\quad \text{between } f_p \text{ and } f, \text{ where } \texttt{Level}(f_p) < \texttt{Level}(f) \end{aligned}$$

where `Level` returns the level value of that node in the graph and the fragment node representing the RF fragment has level value 1 and fragment nodes representing VF fragments are at the lowest level in the graph. As outlined by Equation (6.3) and Equation (6.4), the cost of an operation node is equal to the evaluation cost of the operator represented by the operation node plus the cost of the fragment at the higher level of the operation node in the global view graph; and the cost of a fragment node is equal to the cost of the operation node at the higher level of the context fragment node within the global view graph. The cost of a RF fragment is always 1 and SF fragments are excluded from all computations as they can never be materialised. For example, the fragment cost of VF1 shown in Table 6.1 is computed by accumulating the cost of every node on the path from RF to VF1 in the XFM graph in Figure 4.5. The fragment and operation nodes involved in computing `cost`(VF1) are RF, DF1, DF2, DF3, FF7, FF1, FF2, DF4, FF3, DF5 and all the operation nodes between them.

Additionally, Equation (6.5) is used when computing a fragment f based on another fragment f'. For instance, in Figure 4.5, `cost`(DF3,FF7) returns the fragment cost of FF7 based on DF3, that is the cost of executing the operation node ($\sigma_{//\text{year}\geq 2010}$) over DF3. In such a case, we say that the fragment cost of FF7 is *computed based on* DF3.

Global Benefit. The term Global Benefit is used to measure the contribution of a fragment across all existing views. The global benefit is obtained by using Equation (6.6).

- Global Benefit of a fragment f (`GBenefit`(f)): is the benefit of f that contributes

to the overall view graph, where

$$\texttt{GBenefit}(f) \;=\; \frac{\texttt{share}(f)}{|\mathbb{V}|} \tag{6.6}$$

$|\mathbb{V}|$ is the number of views in the XFM view graph. The global benefit of a fragment is always in the range (0,1], where the global benefit of a RF fragment is always 1 and the global benefit of all VF fragments is close to 0. The larger the global benefit, the greater the chance that this fragment can contribute to other fragments (views).

6.2.1 Cost Matrix

Table 6.1 on Page 90 demonstrates the estimated costs of sample fragments within the global view graph shown in Figure 4.5. As will be shown in our experiment chapter, the Worldbikes dataset contains 4.15GB data with 380,275,484 nodes in total, 8-level depth and the usage of each fragment is computed using the user preferences that are generated randomly in our experiment. For ease of understanding the costs in Table 6.1, we now briefly explain how they were calculated.

- Shareability Column: the number of views in the global view graph sharing the context fragment, e.g., in Table 6.1, `share`(DF1) = 5, as there are five views in the global XFM view graph sharing the fragment DF1.

- Maintenance Cost Column: the number of nodes that are actually stored. The maintenance cost of a context fragment is treated as the number of instance nodes within the fragment and all of the nodes within their subtrees. For example, in Table 6.1, `maincost`(FF7) = 27222192, that is there are 27,222,192 nodes within the subtree of all instance nodes of FF7.

- Reuse Cost Column: the value of the reuse cost is equal to the average fragment costs based on the context fragment, e.g., the fragment costs of FF1 and FF8 are computed based on FF7. The fragment costs of those fragments are computed using Equation (6.5) and the reuse cost of the context fragment is then calculated using the

formula listed below:

$$\texttt{reusecost}(f) \quad = \quad \frac{\texttt{cost}(f, f_1) + \texttt{cost}(f, f_2) + \cdots + \texttt{cost}(f, f_n)}{n} \quad (6.7)$$

,where $\texttt{Level}(f_i)$=$\texttt{Level}(f)$+2 and $0 < i \leq n$.

Equation (6.7) outlines the fact that the reuse cost of the context fragment f is the average cost of the fragment costs of $f_1 \cdots f_n$ computed based on f. For instance, as shown in Figure 4.5, the reuse cost of FF7 as follows:

$$\texttt{reusecost}(\text{FF7}) \quad = \quad \frac{\texttt{cost}(\text{FF7}, \text{FF1}) + \texttt{cost}(\text{FF7}, \text{FF8})}{2} = 39048.$$

- Materialisation Cost Column: This is the cost of the context fragment plus the network communication cost. We consider the evaluation cost as the fragment cost of the context fragment and the number of nodes required to be transferred over the network as the network communication cost. Bear in mind, the network communication cost is only considered when source data and views are stored on different sites over the network, which is the context for our research.

- Cardinality Column: the number of instance nodes in the context fragment. During the XML document parsing process, we count the number of instance nodes mapped to each node in the SchemaGuide and store these statistics in the SchemaGuide. Bear in mind that we are dealing with XPath queries and thus, there may be more than one part of the SchemaGuide that matches the query. For example, `/Project/Name` may occur in different contexts throughout the SchemaGuide. In this case, we sum up the instance counts for each part of the SchemaGuide.

- Fragment Usage Column: We randomly generate the usage of each view, and then, based on the view usage, we compute the fragment usage using Equation (6.1). This has the effect of possibly adding to the set of materialised fragments and is crucial where fragments are not be selected based on the estimated costs, but have a high frequency of usage in real world situations.

- Frequency Column: computing the frequency of the context fragment is straightfor-

ward by applying Equation (6.2) according to the fragment usages.

- Fragment Cost Column: the value of the fragment cost is computed by accumulating costs of all fragment and operation nodes along the path from the RF fragment to the context fragment. As part of this, it is necessary to estimate a cost for each operator. However, this cost is determined by the underlying database technology. For example, we use MonetDB which has one set of operators costs while Oracle or SQL Server will have another set of costs. As our approach is technology independent, we assign a fixed constant to this cost (currently set at "1") for all operators, i.e., they are of equal cost. As shown in Table 6.1, the fragment cost of VF1 is far bigger than that of the other fragments as the fragment cost starts from the root, and VF1 is at the end of the longest path.

- Global Benefit: the value of the global benefit is computed using Equation (6.6), that is the result of dividing the shareability of the context fragment by the total number of views in the global XFM view graph. For example, as shown in Table 6.1, the global benefit of DF1 is 1 as it is shared by 5 views and there are total number of 5 views in the graph, therefore, `GBenefit`(DF1) = $\frac{5}{5}$ = 1. On the other hand, the global benefit of FF2 is 0.8 as there are only 4 views out of 5 sharing it, thus, `GBenefit`(FF2) = $\frac{4}{5}$ = 0.8.

6.3 Cost-Based Greedy Heuristics

In this section, we present a set of cost-based heuristics to drive the filtering process for fragments. There are five heuristics in all, each using different combination of costs.

Size Heuristic : The maintenance cost of a selected fragment should be less than a predefined threshold M, $\texttt{maincost}(f) < M$.

The *Size Heuristic* is concerned with the materialisation size as one does not want to materialise a fragment containing the number of instance nodes that is more than a certain percentage of the original data size. Based on empirical studies, we have set this threshold at 0.08 or 8% of the overall size of the dataset in our implementation. This heuristic uses

column Maintenance Cost in the matrix to determine the size.

$$\textit{Frequency Heuristic} \quad : \quad \texttt{freq}(f) > F, \text{ where } 0 < F \leq 1.$$

In addition to the maintenance restriction specified in the *Size Heuristic*, the process also ensures that the frequency of a fragment exceeds a predefined rate. Unlike the *Size Heuristic*, which focuses on the storage requirement of a fragment, the *Frequency Heuristic* concentrates on the usage of fragments of existing views (columns Fragment Usage and Frequency in the matrix). The Frequency Heuristic states that the frequency of a fragment should be greater than a threshold F, where F is a predefined value. The motivation is not to materialise a fragment that is used only in a single view or even those with a relatively low frequency rate. During our experiments, we discovered that by setting the threshold F to be the average frequency of the overall fragments, we generally obtained the best selection plan.

$$\begin{aligned} \textit{Reuse Heuristic} \quad : \quad & \texttt{cost}(f) + \texttt{matcost}(f) + \texttt{reusecost}(f) * (\texttt{share}(f)\text{-}1) \\ & < \texttt{cost}(f) * \texttt{share}(f) \end{aligned}$$

The *Reuse Heuristic* determines that for each fragment within the view graph, its materialisation and reuse cost should be less than the cost of computing it from scratch for every change. Columns Shareability, Reuse Cost, Materialisation Cost and Fragment Cost in cost matrix are used to make the determination.

Through observation, one can see the contradiction or tradeoff between the *Reuse* and *Frequency* heuristics. For instance, a fragment may have very high usage (or frequency rate), but still failed to pass *Frequency Heuristic* due to its high execution, materialisation and reuse cost (see the left operand in *Reuse Heuristic*). To avoid this occurring, we extend the *Reuse Heuristic* to take user preferences into account. The concept is that based on the frequency rate of a fragment f, the left operand of the *Reuse Heuristic* should be reduced by a factor of n_f, where n_f = 1-$\texttt{freq}(f)$, $\texttt{freq}(f) \in (0, 1]$, to reflect its user preference.

Therefore, the higher the frequency rate, the less the cost will be. The *Extended Reuse Heuristic* listed below uses the Usage column from the cost matrix.

Extended Reuse Heuristic : $(\texttt{cost}(f) + \texttt{matcost}(f) + \texttt{reusecost}(f) * (\texttt{share}(f)\text{-}1)) * n_f < \texttt{cost}(f) * \texttt{share}(f)$

The cost-based heuristics defined so far ensure that selected fragments have better reusability, lower maintenance cost and relatively high frequency rate. However, one must also consider the benefit of a fragment contributing to the overall view graph. The global benefit of a fragment (column Global Benefit) must exceed a predefined threshold. In our evaluation, we set the threshold value to be the *average* global benefit value of all views.

Global Heuristic : The global benefit of a fragment must exceed the threshold G, $\texttt{GBenefit}(f) > G$.

The cost estimation heuristics presented in this section are loosely based on [RSSB00, Bel04]. However, it was necessary to extensively modify these approaches due to different characteristics of relational and XML data models. [RSSB00] focus on multi-query optimisation, which addresses the problem of optimising sets of queries that may have common sub-expressions based on different cost-based heuristics. Any common expression satisfying all cost-based heuristics is materialised for further query processing. On the other hand, [Bel04] provides two notions of measures: *local benefit* and *global benefit*, for fragment selection. The selection is made based on the local and global contribution of a fragment to existing views. The cost-estimation mechanism for our XFM view framework uses both approaches, but it was necessary to develop new cost heuristics suitable for XML data.

6.4 Fragment Selection Mechanism

In this section, we give a detailed description of the Fragment Selection and View Coverage processes, particularly the Graph-Based, Clustering-Based and Subgraph-Based selection mechanisms involved in these two processes (see Figure 6.1). Graph-based selection per-

forms an *overall* scan over the entire graph to select fragments for materialisation, whereas clustering-based and subgraph-based approaches divide the view graph into small groups (a subset of views) and select fragments based on those groups.

6.4.1 Fragment Selection

During the Fragment Selection process, an overall scan is performed over the global (Full) XFM graph and the cost matrix is generated in the context of the entire view graph, e.g., shareability is computed corresponding to the total number of views in the XFM view graph. We refer to this type of selection as the Graph-Based selection as the cost matrix is calculated based on the entire view graph, which differs from the other two types of selection mechanisms, Cluster-Based and Subgraph-Based selections (see Figure 6.1), where new criteria are used and cost matrixes are computed based on clusters or subgraphs (see §6.4.2). Graph-based selection performs a baseline scan over the entire fragment list to select an initial set of fragments for materialisation. Fragments are filtered by the cost-based heuristics and only those fragments that survive the filtering process are retained for materialisation. The problem with graph-based selection is that the process may drop *good* fragments. This is due to the fact that cost estimation is superior when views within the graph are very similar, i.e., views have most of their sub-expressions shared. There is a high probability that fragments selected by graph-based selection will cover a small number of views. Thus, most views are not materialised and this reduces the possibility that the view adaptation process can reuse fragments to manage view definition changes. Therefore, it is necessary to group views that are similar and perform a "group" based selection to ensure that every view is covered by at least one fragment (this is for the purpose of query processing). The Cluster-Based and Subgraph-Based selections introduced in the next section refine the selection plan by grouping views into different clusters and ensuring that all views are covered.

6.4.2 View Coverage

In the Fragment Selection process, when each view is represented by at least one fragment, the selection process is terminated. However, in the event that some views have no materialised fragments, we proceed to the View Coverage process to address this situation.

In effect, this involves a similar process in that we create a cost matrix and apply the same heuristics to select fragments. However, the construction of the matrix differs with changes to columns Shareability, Reuse Cost, Usage, Frequency and Global Benefit. To effect these new changes, we introduce our clustering strategy with associated similarity measures. In general, this will result in the selection of many more fragments after the application of heuristics. We again check to see if all views are covered. If not, we recompute the cost matrix once more, and apply the heuristics again, possibly several times, until all Views are covered. We begin our discussion with a description of how we compute the new matrix based on clustering and then describe the final iterative process (Subgraph-Based Selection), to complete the materialisation graph.

6.4.2.1 Clustering Based Selection

Although views are already grouped based on their common parts during XFM view graph construction, calculating similarities between arbitrary pairs of views based on their common parts do not adequately distinguish views. To be more precise, one needs to take their length and *uncommon* parts into account to compute their degree of *similarity*. The uncommon parts between views are the sub-expressions that can not be shared between them.
Recall that, a view is represented by a sequence of fragment and operation nodes. To measure the similarity between two views, we can simply compare fragments that form those two views sequentially within the view graph. Inspired by [JW02], which provides a general method to measure the similarity between objects on graph models in the information retrieval context, we have developed our own set of similarity measures that can be applied to XFM view graphs.

$$\texttt{SIM}(V_a, V_b) \quad = \quad (\sum_{i=0,j=0}^{k} \texttt{SIM}(f_{i,a}, f_{j,b}) + \sum_{h=0}^{m} \texttt{LevelDecay}(f_h)\ )/k \qquad (6.8)$$

$$\texttt{LevelDecay}(f_h) = \begin{cases} \texttt{SIM}(f_{k,a}, f_{k,b}) \times C_L, & \text{where } h = 0 \qquad (6.9a) \\ \texttt{LevelDecay}(f_{h-1}) \times C_L, & \text{where } h \geq 1 \qquad (6.9b) \end{cases}$$

$$k = \begin{cases} |V_a|, m = |V_b| - k, f_h \in |\mathbb{F}_b|, & \text{if } |V_a| \leq |V_b| \quad (6.10a) \\ |V_b|, m = |V_a| - k, f_h \in |\mathbb{F}_a|, & \text{if } |V_b| < |V_a| \quad (6.10b) \end{cases}$$

$$SIM(f_{i,a}, f_{j,b}) \begin{cases} 1, & \text{if } f_{i,a} \equiv f_{j,b} \quad (6.11a) \\ \texttt{SIM}(f_{i-1,a}, f_{j-1,b}) \times C_T, & \text{if } f_{i,a} \neq f_{j,b} \quad (6.11b) \end{cases}$$

We define a function SIM, which returns the similarity between two objects, i.e., pairs of views or fragments. Given two views V_a and V_b, $\mathbb{F}_a$ and $\mathbb{F}_b$ are the sets of fragments (SF fragments are not included) within V_a and V_b, respectively. We now present the formulae used to calculate the similarity between views and fragments. The process of measuring the similarity between two views is to compare the corresponding fragments sequentially, starting from the RF fragment and moving towards the VF fragment. Equation (6.8) calculates the similarity between two views based on three steps as listed below:

1. $\sum_{i=0,j=0}^{k} \texttt{SIM}(f_{i,a}, f_{j,b})$: this sums the similarities of all fragment pairs in both views. Each fragment is compared only once with its corresponding fragment in sequential order.

2. $\sum_{h=0}^{m} \texttt{LevelDecay}(f_h)$: where one view contains more fragments than a second view, the "redundant" fragments have no fragment with which to compare. However, rather than ignore them, we assign each "redundant" fragment a *level decay* value to further refine the similarity between those two views. The *level decay* value is necessary where one view is a fragment subset of the other. Without *level decay*, those two views would have their similarity equal to 1, which simply means that they are equivalent. The *level decay* for each "redundant" fragment is equivalent to the level decay value of its previous fragment multiplied by a constant value C_L, which is referred to as the *level dissimilarity decay*. The range of the level dissimilarity decay is in the range of (0,1). By applying the level dissimilarity decay, the process ensures that the similarity between views are more accurate. The sole exception is that for the first "redundant" fragment, its level decay value is obtained by multiplying the similarity of last pair of fragments by the *level dissimilarity decay* (see

View / View	V_1	V_2	V_3	V_4	V_5
V_1	1	0.64	0.5	0.73	0.8
V_2	0.64	1	0.5	0.64	0.7
V_3	0.5	0.5	1	0.5	0.5
V_4	0.73	0.64	0.5	1	0.8
V_5	0.8	0.7	0.5	0.8	1

Table 6.2: Similarity Matrix

Equation (6.9a) and Equation (6.9b)).

3. Calculate the average similarity between fragments: the final step is to calculate the average of the similarity values. We divide the sum of the similarity values and level decay values by the number of fragment pairs (k), see Equation (6.10a) and Equation (6.10b) for the value of k, and the result is the similarity between two views.

Equation (6.11a) and Equation (6.11b) are used to calculate similarities *between fragments*. When two fragments are equivalent, their similarity is equivalent to 1, otherwise, their similarity is equal to the similarity of the previous pair of fragments multiplying a constant value C_T, where $C_T \in (0, 1)$. C_T is referred to as the *term* dissimilarity decay and *term* simply indicates the XPath *NameTest*s mapped to the fragments.

When comparing two fragments, Equation (6.11b) takes the preceding pairs of fragments into account so that the current similarity value reflects not only the similarity between the current pair of fragments, but also the context sub-structure of those two views from the RF fragments to the current fragments.

Example 6.1 (Computing similarity between V_1 and V_5 in Figure 4.5)
Since V_1 and V_5 have common fragments shared from RF to DF4 and differ after DF4, therefore, the similarities between all pairs of fragments from (RF,RF) to (DF4,DF4) are equal to 1. After calculating the similarity of pair (DF4,DF4), FF3 is compared to DF25. As FF3 and DF25 are different, their similarity is calculated based on the similarity of the previous pair of fragments, (DF4,DF4). Therefore, `SIM`*(DF4,DF4) is multiplied by the* term dissimilarity decay *value, C_T. The same steps are applied to the pair DF5 and VF5. However, as VF1 is the "redundant" fragment, having no fragment in V_5 to compare with, we must calculate its* level decay *value, which is equal to the similarity value of its*

previous pair, `SIM`*(DF5,VF5), multiplied by* level dissimilarity decay *constant. Therefore,* `LevelDecay`*(VF1) =* `SIM`*(DF5,VF5)* $\times$ C_L*. Eventually, the similarity between* V_1 *and* V_5 *is computed by dividing the sum of the similarities of all fragments pairs and the level decay values by the number of fragment pairs.* ■

Table 6.2 lists similarities of sample views in the XFM view graph shown in Figure 4.5. By observation, V_1 is similar to V_4 and V_5, relative to its similarity to V_2 and V_3. This is because they share more fragments, i.e., fragments from RF to DF4. V_1 is more similar to V_5 than V_4 as in this case, we have set the value of *term* dissimilarity decay to be 0.1 and the *level* dissimilarity decay to be 0.9 in our system. Thus, we consider two views to be more similar if they differ from each other by the length of the query rather than by different terms (NameTests). By empirical studies, we have found that setting these term and level dissimilarity decay values, leads to a better clustering performance. To achieve this, we purposely generated a set of queries where similarities were known in advance, and examined the similarity values determined by the system.

Creating Clusters. Through clustering, we create a set of subgraphs, where a subgraph consists of n views that are *similar*. Thus, these new subgraphs will result in new cost estimations with the result that a subset of the fragments of the views in each cluster is selected for materialisation. We now describe the process that classifies each view into one of the new clusters.

As the fundamental concept of clustering is to group similar views into the same cluster, the classification process is based on a predefined value called the *similarity threshold*. Views that have their similarity above the *similarity threshold* are stored in the same cluster. Different *similarity thresholds* may lead to a different clustering strategy, which further affects the performance and accuracy of the selection. By deliberately choosing the value of the *similarity threshold*, we can significantly improve the performance of our view adaptation process. As shown below, in our approach, the clusters are generated in an incremental manner by iteratively changing the value of the *similarity threshold* and refining the clusters on the basis of the new *similarity threshold* value.

1. Create a similarity matrix as per Table 6.2. This is highly beneficial due to the many

comparisons between views during clustering.

2. Using the similarity matrix, calculate the average similarity between views, which becomes the initial *similarity threshold*.

3. Cluster views according to the *similarity threshold*. Basically, each cluster has a view as its representative and that is always the first view added to that cluster. If no cluster has already been created, the context view is added into a new cluster, otherwise, the context view is compared to each view representative of existing clusters,

   (a) Take the first view V_1 and add it to the first cluster C_1 (create the first cluster).

   (b) Take the second view V_2 and lookup $\texttt{SIM}(V_1,V_2)$ in the similarity matrix.

   (c) If this value exceeds the *similarity threshold*, then V_2 is added to cluster C_1.

   (d) Otherwise, V_2 is placed in cluster C_2 (creating the new cluster C_2).

   (e) The next view is compared with each cluster representative until the similarity value exceeds the *similarity threshold* or until there is no cluster left, and at that point, a new cluster is formed with the current view as its representative.

   (f) The process continues until all views are placed in clusters.

4. Calculate the average similarity for each cluster. Using these similarities, compute the average similarity across clusters which becomes the new *similarity threshold*. For each cluster, we now determine:

   (a) If the similarity for cluster C_i, where $0 < i \leq n$ and n is the number of clusters, exceeds the current *similarity threshold*, then there is no need to further process C_i, and we say that C_i is "*fixed*". No more views can be added.

   (b) If the similarity for cluster C_i does not exceed the current *similarity threshold*, delete C_i and its views are reprocessed in the next iteration.

5. If all clusters are "*fixed*", then there is no view remaining and the process is then terminated. Otherwise, set the new *similarity threshold* to be the average similarity of the remaining views and start from step 3 again.

By iteratively computing the average similarity for each cluster, we ensure that over time, views that are grouped in the same cluster have highest similarities. The reason for this approach is because if we set a *similarity threshold* to be a value close to 1, there will be many clusters generated containing only a single view. On the other hand, if we set the *similarity threshold* value to be too close to 0, then although the number of clusters generated will be quite small, we will have the same problem as with the original global graph: we will have views not represented and additionally, the lower level of fragment sharing across views will mean higher numbers of fragments selected and thus, a higher view adaptation cost. Therefore, by following the iterative process, we minimise the number of clusters and improve the accuracy of selection.

The goal of the clustering algorithm is to create sub-graphs of similar views which will ensure a good selection plan where most or all views will have fragments selected. We now conclude with a description of this final fragment selection process.

6.4.2.2 Subgraph Based Selection

After clustering, there may still be a number of views for which no fragment has been selected for materialisation. This final process takes all of these views and creates a single graph (a small subgraph of the overall XFM graph), and begins an iterative process of locating the best fragments for selection. The following steps are performed during the process:

1. Based on the current subgraph, create a new cost matrix.
2. Using the heuristics, select new fragments.
3. Remove the views for which fragments have been selected and thus, create a smaller subgraph.
4. If the process reaches a point where an iteration sees no fragment selected, it then selects the VF fragment for the view and terminates the process. Otherwise, go to step 1.

6.5 Summary

In this chapter, we presented our fragment selection strategy for the XFM view graph. As part of this strategy, Graph-Based selection uses costs estimated on the full view graph. It performs well when all views are relatively similar. The Cluster-Based and Subgraph-Based approaches refine the estimated costs in order to select any "missed" fragments within clusters and ensure all views are covered by selecting new fragments from a set of uncovered views (subgraphs).

So far, we have discussed the XFM view graph (Chapter 4), the containment algorithms (Chapter 5) used for building the graph and for detecting the degree of changes between views and in this chapter, the fragment selection was presented. In Chapter 7, we will present the final piece in the XFM View Framework, the view adaptation process, which manages changes to view definitions.

Chapter 7

A Fragment-Based View Adaptation Mechanism

At this point, we have framework, model, and algorithms necessary to maintain our view graph and can now proceed to developing the process for adapting the view graph. This chapter starts by outlining our view adaptation approach in §7.1 and provides a description of some of the basic components. The adaptation process consists of two phases: *structural adaptation* covered in §7.2, and *data adaptation* which is then discussed in §7.3.

7.1 View Adaptation Outline

In our XFM view framework (see Chapter 4), a set of XML views are integrated into a global XFM view graph. A view to which a change applies within the XFM view graph is referred to as the *target view* and the change is referred to as the *target* or *target fragment*. In effect, a fragment is inserted into, modified or deleted from the XFM view graph. We also refer to the views besides the target view in the XFM view graph as the *non-target view*s.

The concept of view adaptation is to maintain the XFM view graph both *structurally* and *physically* in response to changes to existing views. Maintaining views structurally means that the logical structure of the XFM view graph reflects fragments that are common to more than one view, *before* and *after* applying the changes. Making physical updates implies that

the materialised data (if any) associated with the existing fragments should be adapted to reflect any change applied. Based on the effect taking place on the XFM view graph, a corresponding adaptation algorithm is chosen to maintain the view graph in response to the change. One of three algorithms is selected: *Fragment Insertion*, *Fragment Deletion* and *Fragment Modification*. These algorithms vary in detail but will always follow the two broad phases of structural adaptation and data adaptation [LRB10b, LRB11].

1. **Structural Adaptation** – In this phase, the process maintains the logical structure in response to view definition changes. The Structural Adaptation phase is further divided into three components,

   (a) **Fragment Replication** – The main objective of this part is to first identify whether non-target views are affected by the change and then to replicate the fragments that are influenced by the change. This has the effect of separating the target view from those views affected by the change so that there are no longer any shared fragments between the target view and other existing views.

   (b) **Target View Adaptation** – This part performs the actual execution of the change in the target view, e.g., insert a new fragment, delete an old fragment or modify an existing fragment.

   (c) **Fragment Optimisation** – This part checks to see whether the new view resulting from the change, shares common fragments with existing views.

2. **Data Adaptation** – In this phase, the process physically adapts existing materialised data (if any) associated with the fragments. For each fragment that is affected by the change, the View Adaptation process checks whether or not the fragment is materialised and updates the fragment.

For the purpose of clarity, we first introduce three methods that are extensively used in our view adaptation algorithms,

1. `GetNextFragment`, which returns the fragment at next level of the fragment passed as a parameter. The method takes a *view id* and a context *fragment id* as part of its input. For instance, as shown in Figure 4.5 on Page 54, the next fragment of FF7 in V_4 is FF1, i.e., `GetNextFragment`(FF7, 4) = FF1.

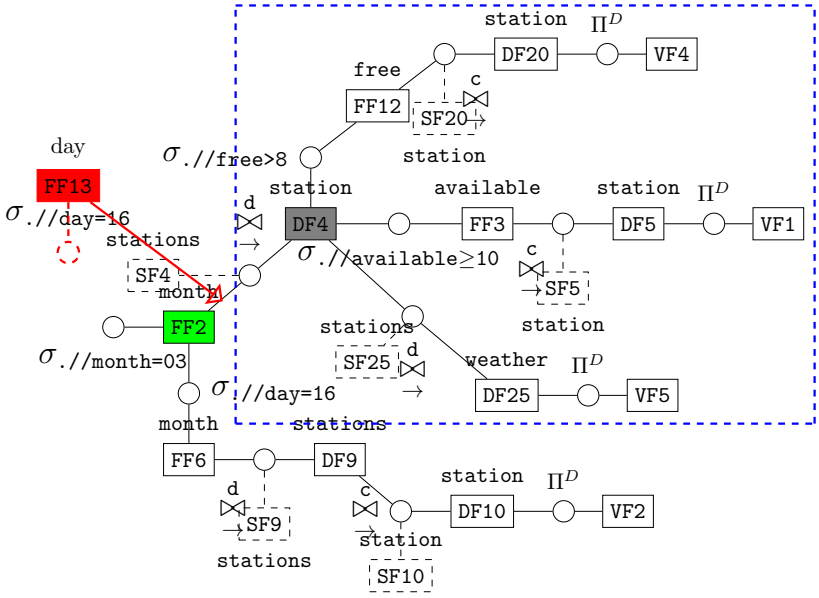


Figure 7.1: Adaptation Area of Fragment Insertion

2. `GetNextFragments`, which returns *all* fragments at next level of a given fragment. Recall that each fragment is assigned a value indicating their level in the XFM view graph. As shown in Figure 4.5, the fragments at next level of FF7 are FF1 and FF8, i.e., `GetNextFragments`(FF7) = {FF8,FF1}.

3. `GetPreviousFragment`, which returns the fragment that is located at the level above the given fragment, or in other words, the fragment preceding the given fragment. For example, the preceding fragment of FF7 in Figure 4.5 is DF3, that is, `GetPreviousFragment`(FF7) = DF3.

It is now useful to identify segment areas in the XFM view graph. Depending on the location where the change is made, we divide the global XFM view graph into two parts: the *Static Area*, where fragments located in this part are not affected by the change; and the *Adaptation Area*, which contains fragments that are update candidates. We refer to fragments in the *static area* as the *static fragment*s (SF) and fragments in the *adaptation area* as the *adaptation fragment*s (AF). The area inside the dashed rectangular box in Figure 7.1 and Figure 7.2 represents the adaptation area. The goal is to identify the segment of the global XFM view graph that is potentially impacted by the change.

Each adaptation process, *insertion*, *deletion* and *modification*, starts by checking the impact

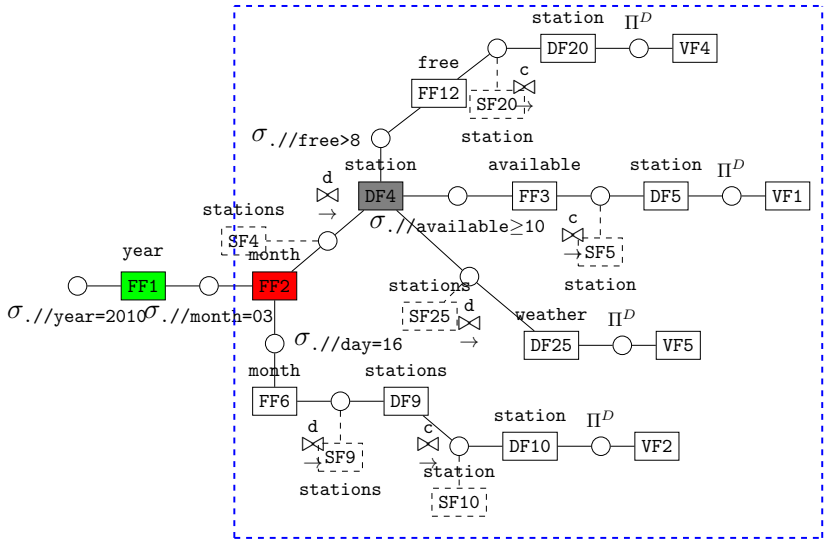


Figure 7.2: Adaptation Area of Fragment Deletion and Modification

of applying a change to the target view. The checking process generally starts from a fragment *preceding* the target fragment with the exception of fragment insertion, where the process starts with the fragment *after* the point that the new fragment is inserted. The fragment that precedes the target fragment is the *Adaptation Area Root*, which we simply refer to as the *Adaptation Root* from now on. As the *Adaptation Root* will never change, it resides in the static area of the graph and thus, provides the link between both areas of the graph.

Figure 7.1 gives an example of adding a new predicate (*.//day=16*) into an existing view after the predicate *.//month=03*. The new fragment (target fragment) is shown as FF13 in Figure 7.1. In this case, fragment FF2 is the *adaptation root* of the insertion process as the new fragment is inserted after it. For *Fragment Deletion* and *Fragment Modification*, as shown in Figure 7.2, assuming that the predicate *.//month=03* must be either deleted or modified, therefore, its corresponding filter fragment, FF2, is either deleted or modified. In this case, the target fragment is FF2 and the adaptation root is the one preceding the target fragment, that is, FF1.

7.2 Structural Adaptation

In this section, we focus on the structural adaptation of our view adaptation mechanism. The structural adaptation phase consists of three components, Fragment Replication, Target View Adaptation and Fragment Optimisation.

7.2.1 Fragment Replication

The basic step required for fragment replication is to iteratively check whether each adaptation fragment on the path toward the VF fragment is shared. If any adaptation fragment is shared between existing views, `FragmentReplication` updates the logical structure of the global XFM view graph so that non-target views retain the same logical structure after applying the changes (*Lines 4-7*, Algorithm 7.1).

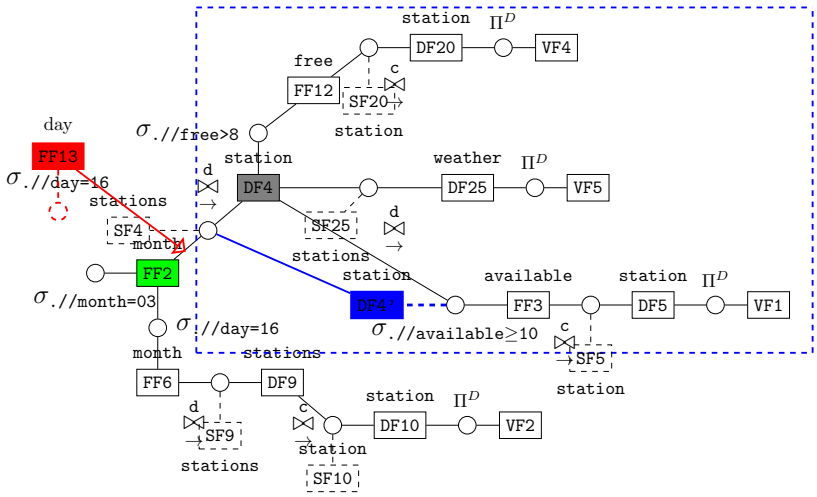
Algorithm 7.1: FragmentReplication(f_{AR}, V, $\mathcal{G}$)

Input: f_{AR}, the *adaptation root*; the target view V; the global XFM view graph $\mathcal{G}$;
Output: an updated version of $\mathcal{G}$;

```
1 f_AF ← GetNextFragment(f_AR, V);
2 if f_AF is not shared then return G;
3 else
4     while f_AF is shared do
5         f_copy ← create a copy of f_AF;
6         replace f_AF with f_copy in V;
7         f_AF ← GetNextFragment(f_AF, V);
8     return G;
```

The algorithm iteratively checks all adaptation fragments toward the VF fragment of the target view and each adaptation fragment is replicated if it is shared (*Line 5*). The adaptation fragment referenced by the non-target views is replaced with the copy (*Line 6*). If the original fragment is materialised, then a copy of the materialisation is also replicated. The process stops when a fragment encountered is referenced only by the target view as once this happens, sharing can no longer occur on the path to the VF fragment. Recall that fragments located at lower levels of the global XFM view graph inherit the *shareability* from fragments at higher levels of the view graph. Therefore, if a fragment is only referenced by the target view, then all fragments starting from this fragment toward the VF fragment are also referenced only by the target view. Example 7.1 demonstrates how *Fragment Repli-*

cation works when a predicate is applied to an existing view. The example is based on the global XFM view graph shown in Figure 4.5 on Page 54, with Figure 7.1 representing a segment of the global view graph.



V_1 : List operation status of the stations in Dublin that have more than 10 bikes available in March 2005.
//Worldbikes//city/Dublin[.//year=2005][.//month=03]//stations[.//available>10]/station

Figure 7.3: Fragment Replication

Example 7.1 (Fragment replication during the insertion process)
Suppose a new predicate day=16 *is applied to* V_1*, which restricts the view to contain only station information of Dublin on the 16-Mar-2005. As shown in Figure 7.3, the new predicate is represented by the new fragment* FF13 *and is inserted after fragment* FF2*. Since fragment* DF4 *is shared between* V_1*,* V_4 *and* V_5*, therefore, before inserting the new fragment, the process must ensure that* V_4 *and* V_5 *are not influenced by the insertion. As a result, the* `FragmentReplication` *algorithm makes a copy of* DF4*, i.e.,* DF4'*. As shown in Figure 7.3, fragment* FF3 *and all the following fragments are reconnected to* DF4'*.* ■

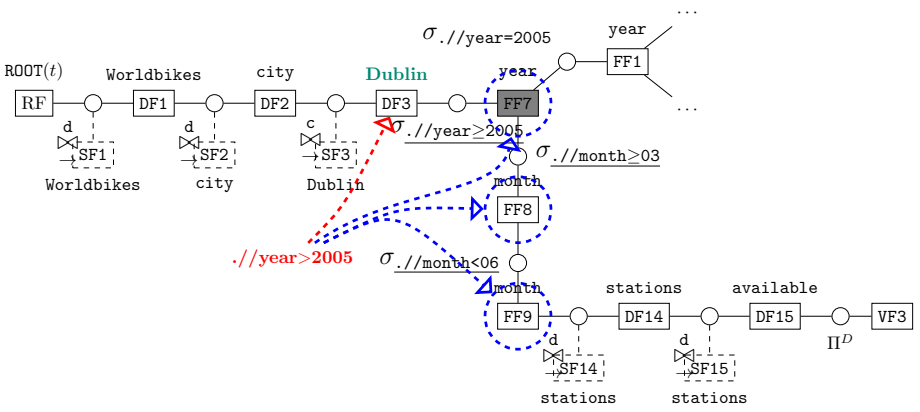
The purpose of *Fragment Replication* is to ensure that when adapting the target view, the adaptation process has no impact on non-target views, and then, in the *Target View Adaptation* sub-phase, the actual structural adaptation is performed.

7.2.2 Target View Adaptation

At this point, the process has isolated the target view from other views affected by the change. The next step is to apply the change to the target view. Depending on the type of the change, either *Fragment Insertion*, *Fragment Deletion* or *Fragment Modification* is performed.

7.2.2.1 Fragment Insertion

The Fragment Insertion process either adds a new step or a predicate to an existing view. In the case of adding a new step, the process merely inserts the corresponding new **Dependency Join Fragment** into the target view. If a new predicate is applied to a step of an existing view, the process needs to check the containment relationship in advance between the new predicate and existing predicates for that step. This is due to the fact that the new predicate may be more or less restricted than the existing predicate. Adding a predicate that is more restricted than an existing one may require an update of the view structure. Based on Figure 7.4, which is a segment of the XFM view graph shown in Figure 4.5, Example 7.2 demonstrates the case when a new predicate is applied to an existing view.



V_3 : List bike availability of each station in Dublin for the months of March, April and May after 2005.
//Worldbikes//city/Dublin[.//year≥2005][.//month≥03][.//month<06]/stations//available

Figure 7.4: Fragment Insertion - Insert A New Predicate

Example 7.2 (Insert a new predicate into V_3 in Figure 7.4)

Assuming that a new predicate year>2005 *is applied to the step* Dublin (DF3) *in* V_3. *Since*

there are other predicates in the view graph also applying to the step Dublin*, the process must compare the new predicate with existing ones. In this case, the predicates represented by the filter fragments* FF7*,* FF8 *and* FF9 *(see predicates with underline) are compared with the new predicate. The location in the global view graph to where the new predicate is inserted depending on the containment relationship between the new fragment and existing filter fragments. As shown in Figure 7.4, the new predicate is less restricted than the existing predicate* year≥2005 *and, as a result, the new predicate is inserted between* DF3 *and* FF7.■

Algorithm 7.2: TargetViewAdaptationForFragmentInsertion(f, f_{AR}, V, $\mathcal{G}$)

Input: the new fragment f to be inserted; f_{AR}, the adaptation root; the target view V and the global XFM view graph $\mathcal{G}$;
Output: an updated version of $\mathcal{G}$;

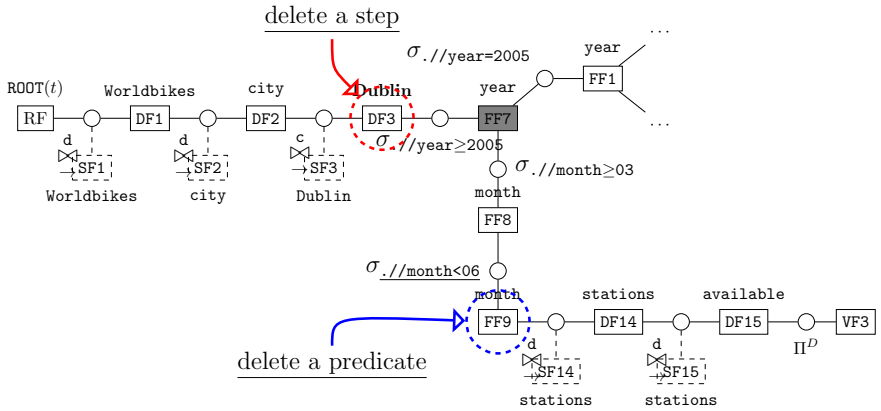
```
1 insert f after f_AR in V;
2 f_next ← GetNextFragment(f, V);
3 while f is a Filter Fragment do
4     if f contains f_next then break;
5     else if f is equal to f_next then remove f_next; break;
6     else if f_next contains f then swap f and f_next;
7     f_next ← GetNextFragment(f, V);
8 return G;
```

As shown in Algorithm 7.2, the fragment insertion process first inserts the target fragment into the target view after the adaptation root (*Line 1*). If the target fragment is a **FF fragment** (*Line 3*), the process then compares it to those **FF fragment**s in the target view, moving *in the direction of* the **VF fragment**. The structure of the target view is updated based on the containment relationship detected between the new fragment and the existing **FF fragment**s referenced by the target view (see *Line 3-7*). Only those filter fragments mapped to the predicates that are applied to the same step as the new predicate are compared by the algorithm.

7.2.2.2 Fragment Deletion

For fragment deletion, there are two possible cases as shown below.

1. If a step is removed, the process also removes all predicates applying to it as they are no longer used to restrict data in the view. This means that the **DF fragment** corre-



V_3 : List bike availability of each station in Dublin for the months of March, April and May after 2005.
//Worldbikes//city/Dublin[.//year≥2005][.//month≥03][.//month<06]/stations//available

Figure 7.5: Fragment Deletion

sponding to the deleted step is removed from the graph together with the predicates (**FF fragment**s) for that step (see *Line 1-3*, Algorithm 7.3).

2. If a predicate is removed, the process merely deletes the corresponding **FF fragment** from the target view (see *Line 4*, Algorithm 7.3).

Algorithm 7.3: TargetViewAdaptationForFragmentDeletion(f, V, $\mathcal{G}$)

Input: the target fragment f; a target view V, and the global XFM view graph $\mathcal{G}$;
Output: an updated version of $\mathcal{G}$;

```
1 if f is a Dependency Fragment then
2 |   delete f from the target view V;
3 |   iteratively delete all Filter Fragments that are applied to f;
4 else delete f from the target view V;
5 return G;
```

Example 7.3 and Example 7.4 demonstrate the cases of deleting a step and deleting a predicate, respectively, based on the segment of the XFM view graph shown in Figure 7.5.

Example 7.3 (Deleting a step of V_3 in Figure 7.5)

Suppose one would like to change V_3 *to display bike availability of stations stored in the database rather than only stations in Dublin. Therefore, the step* /Dublin *is deleted from* V_3 *together with all predicates applying to it. In this case, the fragment corresponding to the step* /Dublin, DF3, *is deleted from the global view graph. Additionally, all fragments*

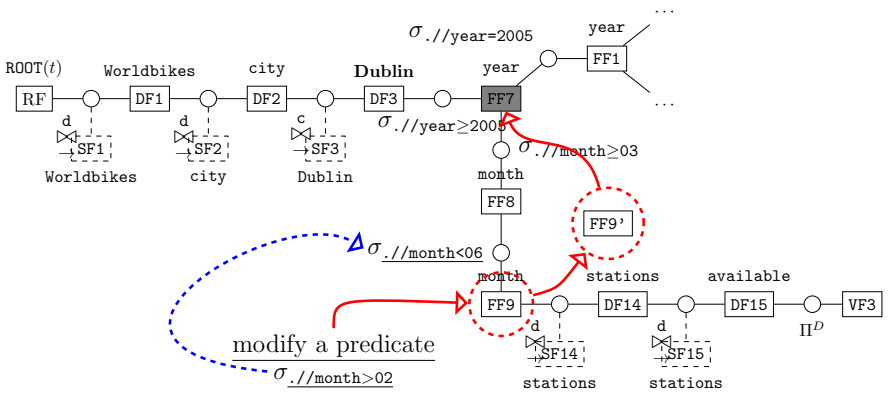
corresponding to the predicates applying to the step /Dublin *are also deleted, i.e.,* FF7*,* FF8 *and* FF9*, as they are no long valid.* ■

Example 7.4 (Deleting a predicate of V_3 in Figure 7.5)
Suppose one would like to change V_3 to list bike availability of stations in Dublin after March, 2005 instead of the months between March and June. For this case, the predicate .//month<06 *is removed. Since the target view has already been isolated from other existing views during fragment replication, fragment* FF9 *corresponds to* .//month<06 *can be safely deleted from the global view graph.* ■

7.2.2.3 Fragment Modification

Fragment modification will always be a result of a change to a view predicate and thus, will affect only filter fragments. After a predicate has been modified, the process checks existing predicates applying to the same step (fragment) to see if the new predicate already exists. If it exists, the modification to the predicate will result in the fragment being deleted. If it does not exist, the modification may result in the repositioning of the filter fragment depending on the containment relationship detected between the existing filter fragments and one resulting from the change. For example, the new predicate should *not* precede an existing predicate that is less restricted than it.

Example 7.5 (Modify a predicate in V_3 in Figure 7.6)
V_3 lists the bike availability of stations in Dublin for the months of March, April and May, after 2005. Suppose one would like to change the predicate from month<06 *to* month>02*, which lists bike availability status in Dublin after February 2005. Since the predicate* month<06 *is mapped to the filter fragment* FF9 *in the graph, changing the predicate from* <06 *to* >02 *causes* FF9 *to be less restricted than* FF8 *(*month≥03*). As a result, the new fragment,* FF9'*, resulting from the modification should be placed between* FF7 *and* FF8 *as shown in Figure 7.6.* ■



V_3 : List bike availability of each station in Dublin for the months of March, April and May after 2005.
//Worldbikes//city/Dublin[.//year≥2005][.//month≥03][.//month<06]/stations//available

Figure 7.6: Fragment Modification

Algorithm 7.4: TargetViewAdaptationForModifyFragment(f, V, $\mathcal{G}$)

Input: a target fragment f; a target view V, and the global XFM view graph $\mathcal{G}$;
Output: an updated version of $\mathcal{G}$;

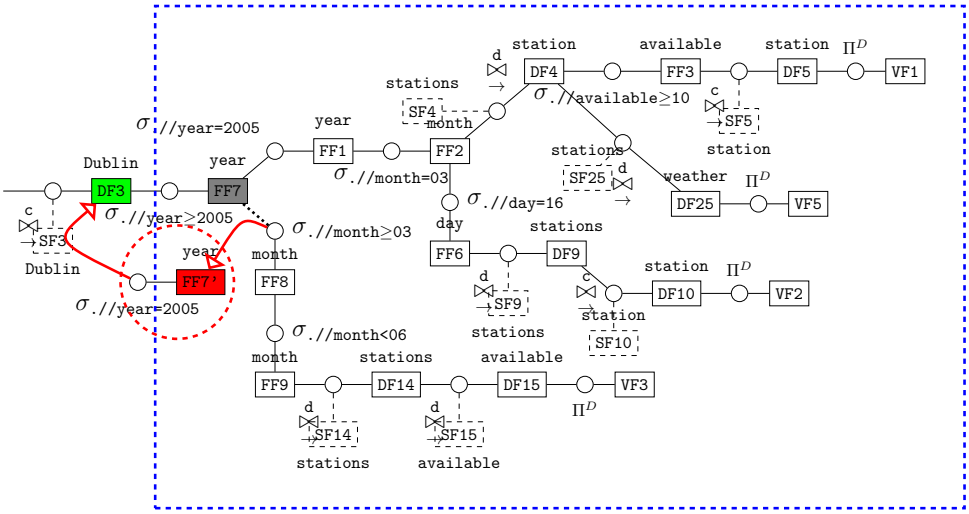
1 $f_{\text{modified}} \leftarrow$ modify f;
2 **while** ($f_{\text{next}} \leftarrow$ `GetNextFragment`(f_{modified}, V)) *is not a* ***VF fragment*** **do**
3 | **if** f_{modified} *equals to* f_{next} **then** remove f_{modified} from V;
4 | **else if** f_{modified} *contains* f_{next} **then** break;
5 | **else if** f_{next} *contains* f_{modified} **then** swap f_{next} and f_{modified};
6 **return** $\mathcal{G}$;

As shown in Algorithm 7.4, the process first modifies the fragment corresponding to the target predicate (*Line 1*). It then checks whether the fragment resulting from the modification conflicts with an existing fragment referenced by the target view (see *Line 2-5*), for instance, whether the modified fragment is equivalent to an existing fragment. As shown on *Line 3* and *Line 5*, respectively, the structure of the target view is updated if the modified fragment is equivalent to, or contains any fragment at next level (towards the VF fragment) of the modified fragment in the target view.

7.2.3 Fragment Optimisation

At this point, we have isolated the target view from previously connected views and have performed the required graph changes. The next step is to optimise the current XFM view graph to ensure that common fragments are still shared. Before discussing the optimisation mechanism, we first introduce the concept of the *Optimisation Area*, which is the area in

the global XFM view graph containing those fragments that could be optimised. In other words, fragments within this optimisation area are potential fragments to be shared. The optimisation area is an extension of the adaptation area as we must include all fragments that may share data with fragments in the target view. These can only be those fragments that reside on paths that extend from the adaptation root. Since the actual change has been performed in the *Target View Adaptation* sub-phase, all fragments within the optimisation area are always more restricted than the adaptation root. Therefore, it is not necessary to check fragments above the level of the adaptation root.



V_3 : List bike availability of each station in Dublin for the months of March, April and May after 2005.
//Worldbikes//city/Dublin[.//year≥2005][.//month≥03][.//month<06]/stations//available

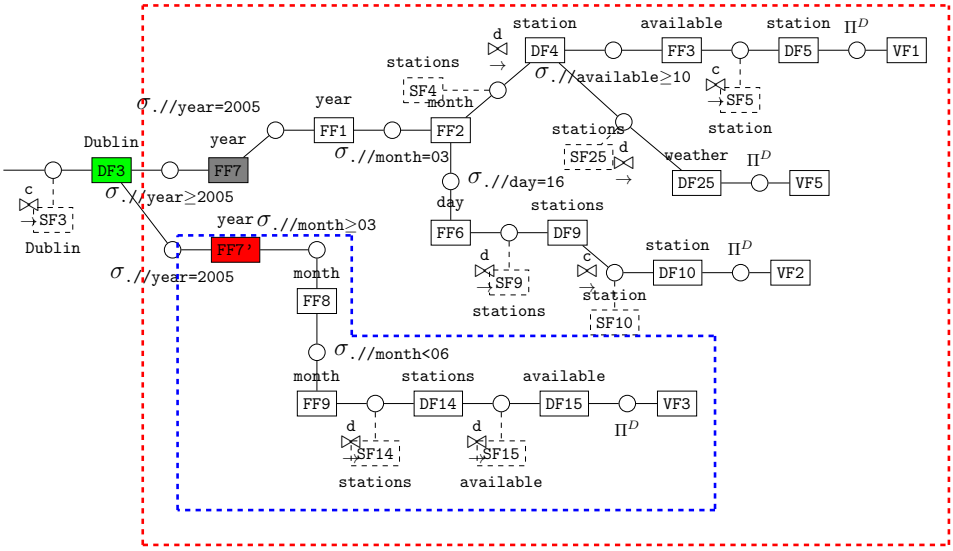
Figure 7.7: Fragment Optimisation - Modifying A Predicate

As demonstrated in Example 7.6, the objective of Fragment Optimisation is to determine if any adaptation fragment can share with any static fragments in the optimisation area.

Example 7.6 (Fragment Optimisation)

Suppose one would like to change V_3 to only list bike availability of stations in Dublin in year 2005 rather than all years after 2005. As shown in Figure 7.7, the predicate year≥2005 *in V_3 is related to the filter fragment* FF7. *Since* FF7 *is shared by other views, a copy of* FF7 *is created by the* `FragmentReplication` *process, i.e.,* FF7', *and* FF7 *is replaced by* FF7' *in V_3. As* FF7 *is materialised, a copy of the materialised data is replicated and referenced by* FF7'. *Figure 7.8 outlines the segment of the XFM graph after changing*

the predicate year≥2005 *to* year=2005. *The area within the dashed box in blue is the adaptation area,* DF3 *is the adaptation root and the dashed box in red is the optimisation area. Through observation, it is not difficult to see that* FF1 *is equivalent to* FF7' *as both represent the result of applying the predicate* year=2005 *to the step* /Dublin. *As a consequence, the* `FragmentOptimisation` *algorithm manages this by comparing static fragments within non-target views after the adaptation root to the adaptation fragments within the adaptation area. As shown in Figure 7.8, as* FF7' *and* FF1 *are equivalent, the* `FragmentOptimisation` *algorithm deletes* FF7' *from the graph and reconnects the rest of the fragments in the target view to* FF1 *(see Figure 7.9 on Page 119) and* FF1 *derives the materialised data from* FF7'. *As will be shown later, as* FF7' *is a copy of the materialised fragment* FF7 *and at this stage the adaptation process concerns only the structural adaptation, the materialised data contained by* FF7' *is still the same as* FF7, year≥2005. ■



V_3 : List bike availability of each station in Dublin for the months of March, April and May after 2005.
//Worldbikes//city/Dublin[.//year≥2005][.//month≥03][.//month<06]/stations//available

Figure 7.8: Fragment Optimisation - After Modification

In summary, as shown in Algorithm 7.5 on Page 120, the process iteratively checks the containment relationship between adaptation fragments (f_{AF}) and static fragments (f_{SF}) next to the adaptation root (going towords the VF fragment in the target view). The process starts from the adaptation fragment (f_{AF}) at next level of the *adaptation root* in the target

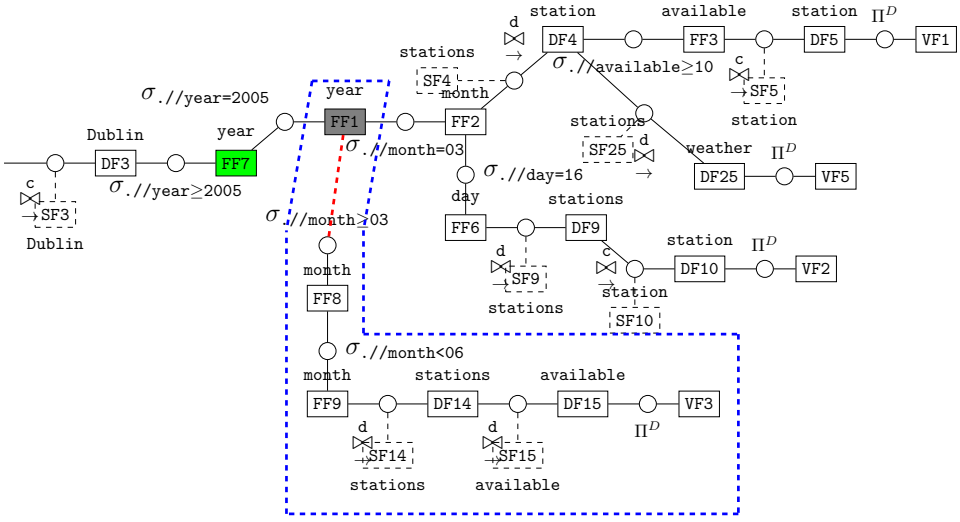
view. There are four possible cases:

Case 1: f_{SF} is equivalent to f_{AF} (see *Line 3*);

Case 2: f_{SF} contains f_{AF} (see *Line 6*);

Case 3: f_{AF} contains f_{SF} (see *Line 10*);

Case 4: no relationship is detected between f_{AF} and f_{SF} (see *Line 13*);



V_3 : List bike availability of each station in Dublin for the months of March, April and May after 2005.
//Worldbikes//city/Dublin[.//year≥2005][.//month≥03][.//month<06]/stations//available

Figure 7.9: Fragment Optimisation - After Optimisation

- In Case 1, if f_{SF} and f_{AF} are equivalent, the process then replaces f_{SF} with f_{AF} (*Line 4*) and continues to check the fragments at next level of f_{SF} in $\mathcal{G}$ (*Line 5*).

- In Case 2, if f_{SF} contains f_{AF} the process reconnects f_{AF} with f_{SF} (*Line 7-8*) fragments at the next level. However, there is still the possibility that f_{AF} may be contained by the fragments at next level of f_{SF} and, therefore, the algorithm recursively calls `FragmentOptimisation` for further containment checking (see *Line 9*).

- In Case 3, if f_{AF} contains f_{SF} the process appends f_{SF} to f_{AF}. For the same reason as depicted in Case 2, fragments at next level of f_{AF} may also contain f_{SF} and, therefore, the process continuously checks the containment relationship between f_{SF} and fragments at next level of f_{AF} in the target view (see *Line 12*).

Algorithm 7.5: FragmentOptimisation(f_{AR}, V, $\mathcal{G}$)

Input: f_{AR}, the adaptation root; the target view V and the global XFM view graph $\mathcal{G}$
Output: an updated version of $\mathcal{G}$

```
1  f_AF ← GetNextFragment(f_AR, V);
2  foreach f_SF ∈ GetNextFragments(f_AR) do
3      if f_SF ≡ f_AF then
4          replace f_AF in V with f_SF;
5          return FragmentOptimisation(f_SF, V, G);
6      else if f_SF contains f_AF then
7          remove the path from f_AF towards the VF fragment in V;
8          append f_AF after f_SF;
9          return FragmentOptimisation(f_SF, V, G);
10     else if f_AF contains f_SF then
11         reconnect f_SF after f_AF;
12         return FragmentOptimisation(f_AF, V, G);
13     else return G;
```

- In Case 4, if no relationship is found between f_{SF} and f_{AF}, then the XFM view graph $\mathcal{G}$ is remain untouched (*Line 13*).

At this stage, all fragments are shared between views in the XFM view graph. The adaptation area is reconfigured with fragments that may be materialised and thus, need to be adapted. As shown in Figure 7.9, the dashed box is the new adaptation area and the adaptation root becomes FF7. In the next section, we show how data adaptation is achieved within the adaptation area.

7.3 Data Adaptation

The second phase, *Data Adaptation*, is responsible for managing the data updates. The main objective is to reuse existing materialised fragments to adapt any fragments that are affected by the change. Recall that after the Fragment Optimisation sub-phase, all fragments are now shared. The potential fragments to be updated are those referenced by the target view and within the adaptation area. As shown in Algorithm 7.6 on Page 121, the data adaptation process involves two searches: i) the *Impact Search* seeks for any materialised fragments in the adaptation area that are affected by the change (see *Line 14-17*); ii) the *Reusability Search* looks for any materialised fragment in the static area that can be reused to adapt

materialised fragments detected in the first search process (see *Line 1-13*). We refer to the materialised fragment that is used for adaptation as the *Reusable Fragment* and both searches start at *adaptation root*. Continuing with the Example 7.6, the search starts from the adaptation root FF7 (see Figure 7.9).

Algorithm 7.6: AdaptFragment(f_{AR}, f_{AF}, V)

Input: adaptation root f_{AR}, a potential effected adaptation fragment f_{AF} and the target view V

Output: A boolean value, true for successfully adapted, otherwise, false

```
1  if f_AF is materialised then
2      if f_AR is materialised then
3          adapt f_AF using f_AR;
4          if f_AF is not the VF fragment then
5              f_AF ← NextFragment(f_AF, V);
6              return AdaptFragment(f_AR, f_AF, V)
7          return true;
8      else
9          f_AR ← PreviousFragment(f_AR);
10         if f_AR is Root Fragment then
11             adapt f_AF;
12             return true;
13         return AdaptFragment(f_AR, f_AF, V);
14 else
15     f_AF ← NextFragment(f_AF,V);
16     if f_AF is null then return false;
17     return AdaptFragment(f_AR, f_AF, V);
```

As shown in Algorithm 7.6, `AdaptFragment` accepts two fragments as parameters. The first input fragment is the adaptation root f_{AR}, which is considered to be a potential materialised fragment that can be reused for data adaptation. The *Reusability Search* starts by comparing f_{AR} to other static fragments on the path toward the RF fragment. This is due to the fact that f_{AR} contains the most restricted data and therefore, the data adaptation process starts from f_{AR} all the way to the RF fragment until a materialised fragment is found. The second (fragment) parameter is a candidate affected by the view redefinition and requires an update. Any fragment located in the adaptation area is a candidate. The *Impact Search* starts from the first fragment in the adaptation area, continuing to the VF fragment in the adaptation area. For example, FF1 in Figure 7.9 is the first adaptation fragment to be processed. The *Impact Search* collects information concerning all fragments that are

materialised and affected by the change for further processing.

As shown in *Line 6* and *Line 17* in Algorithm 7.6, `AdaptFragment` recursively retrieves any adaptation fragments that are "marked" as materialised. As soon as a fragment is found (*Line 1*, Algorithm 7.6), `AdaptFragment` searches for the fragment with the most restricted data (smallest fragment), in the static area that can be reused for the adaptation. By using smaller fragments, we ensure faster adaptation. If this fragment exists, it is used to rematerialise the affected fragment (*Line 3*, Algorithm 7.6). Otherwise, the process continuously searches for a fragment that can be reused, until the RF fragment is reached (*Line 7*, Algorithm 7.6).

Example 7.7 (Searching for impacted and reusable fragments)
As shown in Figure 7.9, the Impact Search starts from the adaptation root FF7 *(exclusive) to the VF fragment of the target view (*VF3*). In this case,* FF1 *is affected by the adaptation process. After the affected fragment is detected, the process then searches for an existing fragment that can be reused with, the Reusability Search also starting at the adaptation root* FF7 *(inclusive) towards the Root Fragment. In this case,* FF7 *is the candidate that can be reused by the adaptation methods introduced in §7.3.1.* ■

7.3.1 Data Adaptation Methods

So far, we have shown how affected fragments and potential reusable materialised fragment are detected. In this section, we show the actual methods used for data adaptation. We consider a distributed environment where views and source data are stored on different sites over the network. Depending on whether data can be obtained locally or remotely, different methods are used. Basically, we classify the data adaptation methods into two types, **Incremental Adaptation** and **Recomputing**, where for *Incremental Adaptation*, the result of the new view definition is obtained by either removing redundant data from the affected fragments or adding extra data into the affected fragments. When redundant data must be deleted, the affected fragments are considered to be *self-maintainable* as data adaptation is achieved by executing an additional query over the affected fragments. However, when it comes to the case of adding extra data, the process must obtain extra data from

either an existing materialised fragment or database servers where source data are stored, where the former one saves network communication cost. For the case of *Recomputing*, to keep the affected fragments updated, the process executes the new definition resulting from the change over either an existing materialised fragment or data sources. Based on different contexts, we refine the *Incremental Adaptation* and *Recomputing* methods into the following classification,

1. **Self-Maintained Incremental Adaptation (IASelf)**: where no extra data is required, views can be adapted by directly removing redundant data from the affected fragment.

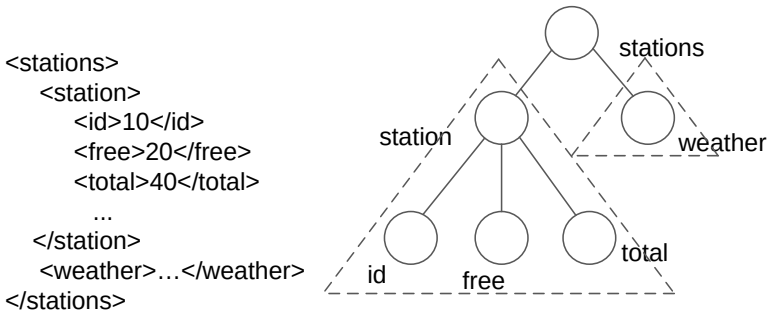
2. **Incremental Adaptation Using Local Fragments (IALocal)**: where extra data is obtained from an existing materialised fragment and then inserted into the affected fragment.

3. **Incremental Adaptation Using Remote Data (IARemote)**: where extra data is obtained from the database servers, transferred to the site where views are stored and then inserted to the affected fragment.

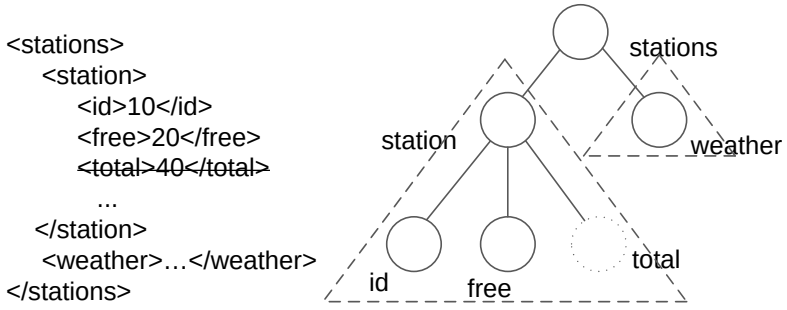
4. **Recomputing Locally (RELocal)**: for this case, the incremental adaptation can not be achieved either because the incremental adaptation is too expensive or because it is not possible to do so. Therefore, this method computes the new definition based on an existing materialised fragment. For example, in Figure 7.9, FF1 is computed based on FF7.

5. **Recomputing Remotely (RERemote)**: this method simply executes the view definition from scratch by using data stored on the database servers that are distributed over the network.

In reality, a query optimiser determines which method to apply during data adaptation based on their corresponding processing cost. The processing cost required by our adaptation methods vary in different implementations. As recomputing only involves the method to find an existing materialised fragment for reuse, which we have already discussed previously in *Reusability Search*, therefore, in the rest of this section, we focus on how incremental adaptation is achieved. Before we start, we first introduce the concept of a *deep-except* and *deep-union* as discussed in [LLLL05]. The *deep-except* and *deep-union* operators are



(a) Before node "free" is deleted (*//stations*)



(b) After node "free" is deleted (*//stations deep-except //stations//total*)

Figure 7.10: Standard Except and Deep-Except Operators

the extended version of the standard *except* and *union* operators introduced in [W3C10a] and are extensively used in our incremental adaptation methods.

Deep-Except Operator The standard *except* operator works on sequences and returns instance nodes appearing in its first operand (sequence), but not the second operand (sequence). The *deep-except* ($-^D$) operator takes two sequences of instance nodes as input and the results is calculated based on three conditions:

1. When the first operand (sequence) is equal to any instance nodes in the second operand (sequence), or is a descendant of any nodes in the second operand, then the result is an empty sequence;

2. When any node(s) of the second operand (sequence) is descendant(s) of the first operand (sequence), it is removed from the first operand. This requires that each instance node within the first operand maintains a reference to their descendants. Therefore, when deletion is necessary to take place, the target instance nodes are lo-

cated by following the node references of instance nodes within first input sequence;

3. Otherwise, when there is no overlap between the first and second operands, the first operand (sequence) is returned.

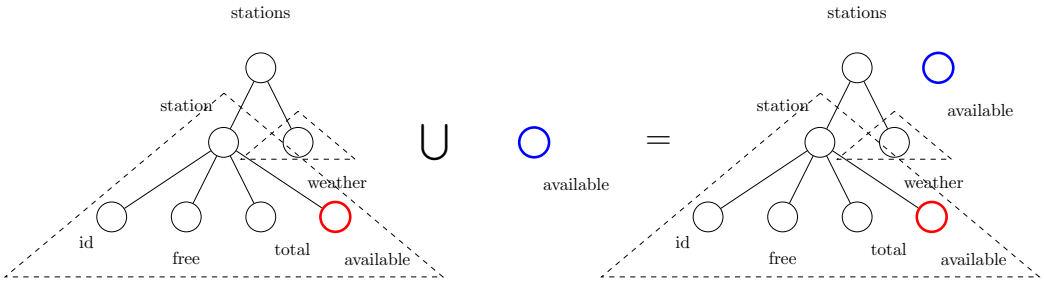
Example 7.8 demonstrates the difference between standard *except* operator and *deep-except* operator.

Example 7.8 (Standard *except* and *deep-except* operators)
Figure 7.10a: //stations
Figure 7.10b: //stations deep-except //stations//total
Result: total *is deleted.* ■

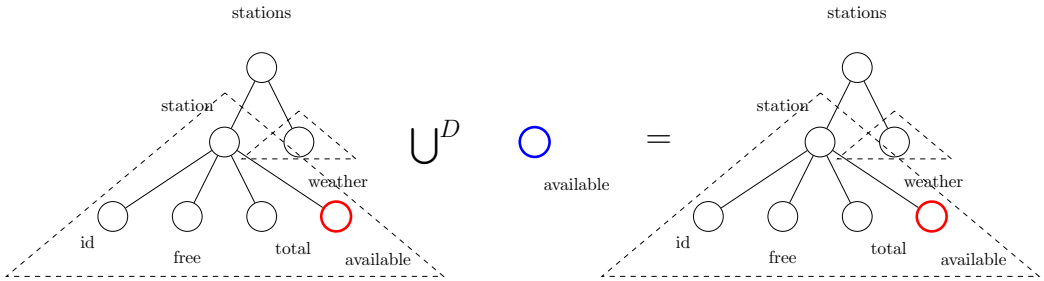
Figure 7.10a demonstrates a segment of the result obtained by executing the query *//stations* over the Worldbikes dataset. By executing the query expression *//stations deep-except //stations//total* over the XML tree in Figure 7.10a, as shown in Figure 7.10b, the node *total* is deleted from the XML tree. If, in Example 7.8, the *deep-except* is replaced by the standard *except* operator, that is, *//stations except //stations//total*, then the result is equal to the segment of the Worldbikes dataset shown in Figure 7.10a. This is due to the fact that *//stations* returns a sequence of *stations* nodes, where as the expression *//stations//total* returns a sequence of *total* nodes. The standard *except* operator returns nodes only in the first sequence, but not in the second. Therefore, for this case, the sequence of *stations* nodes are returned. The *except* operator is not concerned about the instance nodes within the subtree of each *stations* nodes.

Deep-Union Operator The standard *union* ($\cup$) operator takes two sequences as input and returns a sequence containing all the nodes that occur in either of the operands (sequences). Unlike the standard *union* operator, the *deep-union* ($\cup^D$) operator takes two sequences of instance nodes as input and compares not only instance nodes in each sequence, but also their descendants. The deep-union operator returns a sequence of instance nodes which satisfy the two conditions:

1. the returned instance nodes must originate from one of the operands;



(a) Standard Union Operation (*//stations union //station/available*)



(b) Deep-Union Operation (*//stations deep-union //station/available*)

Figure 7.11: Standard Union and Deep-Union Operators

2. the returned instance nodes must not exist in the subtree of a node within either operand.

To successfully perform the *deep-union* operation, it is necessary that each instance node within the operands maintains references to all their descendants.

Example 7.9 (Standard *union* and *deep-union* operators)
Figure 7.11a: //stations union //station/available
Figure 7.11b: //stations deep-union //station/available
Result: stations *is returned.* ■

The sample query of Example 7.9, *//stations union //station/available*, yields a sequence containing nodes *stations* and *available*. If one were to replace the standard *except* operator with the *deep-union* operator, i.e., *//stations deep-union //station/available*, the process would return a sequence containing only *stations* node, the node *available* is already in the subtree of the *stations* node as shown in Figure 7.11b.
We now continue our discussion of data adaptation. For all types of changes, as listed below, there are three possible actions to effect the change:

1. Adding extra data (instance nodes), which involves changes of deleting a step, deleting a predicate and modifying a predicate.

2. Deleting redundant data (instance nodes), which involves changes of adding a step, adding a predicate and modifying a predicate

3. When both 1 and 2 must take place. This occurs in some instances when we modify a predicate.

Assume that the original fragment is f_{old} and the new fragment resulting from the change is f_{new}. We define f^+ as the extra data to be added and f^- as the redundant data to be removed, the data adaptation problem can be expressed as the following equation,

$$f_{\text{new}} \quad = \quad f_{\text{old}} \cup^D f^+ -^D f^- \tag{7.1}$$

Equation (7.1) indicates that the new materialisation (f_{new}) is obtained by adding the extra data (f^+) into the old materialisation (f_{old}) using the *deep-union* operator ($\cup^D$) and then deleting the redundant data (f^-) with the *deep-except* operation ($-^D$). As discussed below, depending on the type of the change being processed, either f^+ or f^- could be empty. meaning we need only add extra data *or* remove redundant data.

Adding Extra Instance Nodes If a change causes the affected fragments to be less restricted, i.e., delete a step, delete a predicate or modify a predicate, as a result, "extra" instance nodes are added into the affected fragments. We must first identify extra instance nodes to be added and then inserting those instance nodes into the affected fragments. To identify extra instance nodes, Equation (7.2) is applied,

$$f^+ \quad = \quad Expr_{\text{new}} -^D Expr_{\text{old}} \tag{7.2}$$

In Equation (7.2), $Expr_{\text{new}}$ is the original expression associated with the affected fragment and $Expr_{\text{old}}$ is the expression mapped to the affected fragment after applying the change. In the case of adding extra nodes, f^- is empty as no instance node is deleted from the old materialisation. The next step is to add instance nodes into the affected fragment. The challenge here is that, by definition, the process must maintain the original order between

instance nodes to be inserted and existing nodes. Our solution is to make use of the position encoding schemes as discussed in Chapter 5, which identifies the position of each instance node within the original XML tree. During insertion, the nodes are inserted into their corresponding *region* (see Definition 5.7 on Page 79 for the explanation of *region*).
During the adaptation process, the right operand of Equation (7.2) is evaluated over either an existing materialised fragment (IALocal) or the source data stored on the database servers (IARemote). If the cost of IALocal and IARemote methods are too expensive, the query optimiser can also decide to apply the RELocal method. Nevertheless, if none of the three approaches is applicable as either they are too expensive or no existing materialised fragment can be reused, the RERemote method must be applied, which recomputes the view from scratch using data stored on the database servers.

Deleting Redundant Instance Nodes Three types of changes restrict materialised data: 1) adding a step, 2) adding a predicate and 3) modifying a predicate. For the case of modifying a predicate, we consider the case that further restricts materialised data. Similar to the previous case, to delete redundant instance nodes, the process needs to first identify the "redundant" instance nodes and then delete them from the affected fragments.

$$f^{-} \quad = \quad Expr_{\text{old}} -^{D} Expr_{\text{new}} \tag{7.3}$$

As shown in Equation (7.3), redundant data is detected by executing the expression specified in the right operand directly over the affected fragment and then deleting them from the affected fragment (IASelf method). Both steps can be achieved by applying an XQuery Update [W3C11] statement as listed below,

$$f_{\text{new}} \quad = \quad \text{do delete } (Expr_{\text{old}} -^{D} Expr_{\text{new}}) \tag{7.4}$$

In the case of deleting redundant nodes, f^{+} is empty as no instance node is added. If the IASelf method is too expensive to achieve, either RELocal or RERmote method is used depending on whether or not there is an existing fragment that can be reused. Different from other cases, the RELocal method used in this case recomputes views based on the affected fragment rather than another existing fragment since the affected fragment already

contains all instance nodes for the new view definition.

Mixture of Adding and Deleting This occurs when a modification causes a dramatic change to the original predicate, e.g., the NameTest is changed or changes take place to the predicates with text values. Our solution to this case is to compute the new view definition over an existing materialised fragment (RELocal). For the case when the affected fragment becomes more restricted, then RELocal is performed upon the affected fragment. Despite the fact that the affected fragment has to be recomputed from scratch, using the existing materialised fragment saves network communication cost required by transferring data from different locations on the network. If no existing materialised fragment can be reused, the RERmote method is used.

7.4 Summary

We now have all the pieces required to develop a system capable of view adaptation for XML views. Furthermore, this system, unlike all other XML view adaptation systems, use a fragment-based approach to deliver increased sharing and improved performance. However, one further step is required in this dissertation. Recall that the hypothesis put forward was that view adaptation is more efficient when using a multi-view based framework with materialised data shared between views. While we have delivered a multi-view adaptation approach, it remains for us to evaluate our system in order to demonstrate that the stated efficiencies can be delivered. In the next chapter, we present a detailed evaluation of both the fragment selection process and the method for view adaptation.

Chapter 8

Evaluating Fragment Based Adaptation

In this chapter, we evaluate the XFM framework and adaptation algorithms. To do this, we implemented the adaptation algorithms on top of two different selection mechanisms: the basic XFM approach (XFM) and an approach using Clustering (XFM-C). We also implemented the Full Materialisation Approach (FULL), which is based on the materialisation of entire views, to show the performance gain and the reduction in materialised data, when using our approach. The FULL approach materialises the query after each change is applied. As the goal is to demonstrate the benefits of the fragment-based framework, we show the percentage of time and storage costs required by XFM-C and XFM relative to FULL approach. We did not apply the single-view based adaptation algorithm proposed in [AML$^+$07] as in their approach, 1) a view is represented by a set of unions and intersections of positive and negative access control rules (XPath expressions), which is quite different to using a single XPath expression; and 2) the result of the view is obtained by performing all deep-union and deep-except operations over all access control rules, which is very time consuming process and is slower than recomputing the entire view. The chapter has two main sections: firstly, we provide a description of the experimental setup with some basic assumptions, and then we proceed to the evaluation and a discussion of the results.

8.1 Experiment Deployment and Implementation

Three database servers were deployed for this experiment: two MonetDB servers act as the remote database servers storing all source XML data, and a third MonetDB server used as a local view repository containing all views and materialised data. This models a typical data warehousing system where data is often distributed due to the high volumes generated. We initially attempted to deploy other native XML databases such as eXist and BaseX but they were unable (or very inefficient) to load large XML datasets, e.g., 4GB, and caused problem when repeatedly creating and deleting views, which are essential in our experiment. We use version 4.38.5 for all MonetDB servers. The remote servers are distributed on two Intel Core(TM)2 Due 2.66Hz workstations running 64-bit Fedora Server 12 with 4GB and 2GB RAM, respectively. The local server is installed on an Intel Core(TM)2 Duo 3.00GHz workstation running 32-bit Windows 7 with 4GB RAM.

Dataset Size (GB)	No. of Elements	No. of Attributes	No. of Text Nodes	Total No. of Nodes	No. of Levels
8.30	405,848,131	2,539,758	351,450,328	759,838,217	8

(a) Worldbikes Data Statistic

SchemaGuide Size (KB)	No. of Schema Nodes	No. of Levels
21.2	388	8

(b) SchemaGuide Statistic

Table 8.1: Worldbikes Data and SchemaGuide Statistics

8.1.1 The Worldbikes Dataset

We opted to use a real world dataset, the Worldbikes dataset (described in Chapter 3), which contains information regarding the Bicycle Sharing System distributed in different cities over the world. For our setup, we stored a Worldbikes dataset (4.15GB each) on each of the remote MonetDB servers, with more than 759 millions of nodes in total and with upto 8 levels as shown in Table 8.1a.

The SchemaGuide. A SchemaGuide for the Worldbikes dataset is created during the parsing process, i.e., using Xerces SAXParser. During parsing, the program incrementally stores each unique combination of root-to-node path and subtree structure of the instance

nodes and a schema node is generated for each combination with an unique identifier (sid). After parsing has completed, the program assigns each schema node a label using the positional encoding scheme, and eventually, stores all schema nodes within a B-tree. The B-tree is small enough to be maintained in the main memory for the purpose of fast access. As shown in Table 8.2, the SchemaGuide created for the two Worldbikes dataset is size of 21.2KB with 388 schema nodes and 8 levels in depth.

8.1.2 Views and Changes

As part of our evaluation, it was necessary to develop a View Generator and a View Adaptation Simulator. To generate meaningful views, valid changes and ensure that views are still meaningful even after applying changes, all paths and value information are retrieved from the SchemaGuide and a Text Value Index of the Worldbikes dataset. For easy access, we store all text values in a hash table, where keys represent names of elements (the node at a higher hierarchy of the tree) containing the text value and the values of the hash table are the actual text values. To avoid storing paragraphs of texts embedded in an XML element, we deliberately store text values by setting the maximum length of the text to be less than 40 characters as in general, it is unusual to have a predicate value more than 40 characters.

par_1	*View Size*	the number of views to be generated
par_2	*% with Predicates*	the percentage of views containing predicates
par_3	*Max. Length*	the maximum length (no. of steps) of a view
par_4	*Common Prefix*	the common sub-expressions shared between views
par_5	*TD Decay*	the term dissimilar decay
par_6	*LD Decay*	the level dissimilar decay

Table 8.2: Query Generator Parameters

8.1.2.1 View Generator.

As shown in Table 8.2, we consider different factors (parameters) when generating views. We control the maximum length of views (*Max. Length*) and the common sub-expressions between views (*Common Prefix*), those two parameters influence similarities between views (or shareability) and the total number of fragments to be generated. The term (*TD Decay*) and level (*LD Decay*) dissimilar decays, control the number of potential clusters that are

View Set	No. of Fragments	No. of Shared Fragments	No. of Steps	No. of Predicates
VS1	2664	368	4133	3513
VS2	2616	337	4099	3472
VS3	2700	362	4121	3529
VS4	2731	349	4120	3552
VS5	2667	366	4116	3506
VS6	2258	309	3525	2376
VS7	3012	357	4022	3139
VS8	2962	348	4091	3093
VS9	2938	379	4056	3120
VS10	2783	338	3797	3105
VS11	2773	367	4252	3167
VS12	2856	365	3870	3029
VS13	2647	351	4251	3036
VS14	2343	323	4473	2437
VS15	2674	358	4251	3063
Avg.	2708	352	4078	3142

Table 8.3: View Statistics

created during the selection.

In our experiment, we have fixed the value of *TD Decay* and *LD Decay* to be 0.1 and 0.9, respectively, as by empirical study, those two values produce far better selection plans. We manually change the *Common Prefix* and *Max. Length* to generate different sets of views which in turn, leads to different adaptation performance.

For an exhaustive evaluation, we generated 15 view sets (15 configurations of the XFM view graphs), where each set contains 1000 views. As shown in Table 8.3, there are an average of 2,708 fragments in each view set and among them, 352 fragments are shared between views. Each view set contains 4,078 XPath steps and 3,142 predicates on average. The View Generator ensures that each step is a valid step relative to its previous step, which is verified by the SchemaGuide to ensure that schema nodes map to the previous and current steps, respectively. If their relationship does not imply the specified relationship (parent-child or ancestor-descendant relationships), the View Generator skips the current step and continues to generate new steps until a valid step satisfying the *axis condition* is obtained. When generating predicates, it first searches for a valid value within the Text Value Index and randomly generates an operator if the type of the value is numeric. The View Generator

compares the new predicate with the existing ones that are applied to the same context (step). If the predicate already exists or it contradicts an existing predicate, e.g., the new one is year>2010 and the existing one is year< 2010, then it is discarded and the process stops if either a valid predicate is generated or specified number of attempts are exceeded. Two sample views generated by the View Generator are given in Example 8.1 and Example 8.2.

Example 8.1 (Sample Query One)
//worldbikes/bikes/city/Dublin[.//year>2008][.//year<2011][.//month=6][.//day=1]
/stations[.//hour=8]/station ■

Example 8.2 (Sample Query Two)
//worldbikes/bikes/city/Dublin[.//year>2007][.//month>=5][.//day=24]//stations
/station/total ■

The query in Example 8.1 simply asks the operational status of each station in Dublin every day after 8 am in June between 2008 and 2011 and the query in Example 8.2 searches for the total number of bikes at stations in Dublin on the 24^{th} of each month after May (inclusive) 2007.

For our experiment, the key factors that determine whether or not clustering is necessary and also the amount of VF fragments to be selected, are the similarity between views and the overall costs of the common fragments between views, e.g., Fragment Cost, Materialisation Cost, Reusing Cost and etc. If a view set contains views that are very similar and with low overall costs for common fragments, then no clustering is required and views can be covered by a small amount of non-VF fragments. If views have either low average similarity value or high overall costs for common fragments, then clustering is required and VF fragments may be selected. Based on these two factors, we demonstrate different circumstances by categorising view sets into three types based on the amount of VF fragments selected by the XFM-C approach and whether clustering is required.

1. Views are very similar to each other and the overall costs of common fragments are low. Thus, no clustering is required and no VF fragment is selected (VS1-VS5).

2. Both the average similarity and overall costs of common fragments are low and as a result, clustering is required. However, no VF fragment is selected (VS6-VS10).

3. The average similarity value is relatively low and the overall costs of common fragments are high. For this case, clustering is performed and View Fragments are selected (VS11-VS15).

For each view set, 40 changes are applied, where each change is guaranteed to be meaningful, and changes are applied in a sequential manner, e.g. adding a predicate followed by adding a step. The changes are randomly created by the View Adaptation Simulator with a variety of change types.

We also randomly generate user preferences for each view. User preferences are of benefit to the user only during query evaluation as this process becomes faster due to the materialisation of the popular fragment. As the focus of this work is on view adaptation (when users change their queries), the effect will generally be negative when a query using a popular fragment is changed. Thus, we randomly generate user preferences to highlight their *impact* on the adaptation process.

8.1.2.2 View Adaptation Simulator.

The type of changes that are allowed includes: 1) add a step or predicate, 2) remove a step or predicate and 3) modify a predicate, which involves making changes to the operator applied to the NameTest (see §4.2 on Page 47), the value of the predicate, or the NameTest to which the predicate applies. An example of the change would be *change year from 2011 to 2001*, or *change year>2007 to year<2010* in Example 8.2.

Changes are pre-generated and stored in an external file. During the evaluation, the simulator reads changes sequentially from the external file and applies them to the global XFM view graph. Changes are generated based on the existing views with the change type randomly selected. An existing view is selected and based on the type of change, the generator traverses the selected view, finds a place to apply the change and then creates all necessary components of the change. For example, if a step is added, the generator first picks an axis (relationship) of the step, searches for the SchemaGuide for a NameTest of the current step and ensures that nodes for the new step, and the step after the position the new step is added,

satisfy the *axis condition*. For the case of adding a predicate, an operator and a text value are also generated, based on the type of the value (numeric, string or date), a corresponding operator is selected. For example, if a text value is a type of either numeric or date, then one of the five different type of operators can be selected ($=$, $\leq$, $\geq$, $>$ and $\neq$). However, if the text value is a string value, then only the $=$ operator can be applied. Furthermore, all text values are selected from the Text Value Index and validated against existing predicates to avoid duplication and contradiction.

8.1.3 Limitations of Current XML Technology

One of the reasons we undertook this research was that despite the growing demand for XML query languages and data warehouse functionality, the underlying technology remains quite slow. As our research requires, to some extent (or at least for evaluation purposes) existing technology, we examined a number of XML database technologies and deemd MonetDB to be among the best. However, efficient updates are still an open research problem for XML researchers and updates remain slow for large datasets. For our experiments, we could not apply the incremental methods introduced in §7.3.1 for two reasons, 1) existing XML databases does not have the required performance for XML updates and it is often the case that recomputing is much faster, and 2) for relational tuple-based data, it is not necessary to retain their orders after adaptation, but for tree-based XML data, structure and order are essential. For example, to add extra instance nodes into a view, the process must determine the original order and hierarchy structure between the new instance nodes and the existing ones. In XML databases, structural and order information between nodes are explored by the underlying labelling/encoding schemes used, i.e., the index. Nevertheless, since our view adaptation system is deployed on top of the MonetDB, we have no access to the underlying index and our implementation of the deep-union and deep-except operators cannot be suitably efficient as they operate on the node level rather than the index level.

8.2 Experiment Evaluation on the XFM Framework

Despite the fact that the incremental methods could not be applied, the fragment-based approach remains far more efficient than the FULL approach as we can still reuse existing

	No. of Clusters		Candidates		No. of VFs		% of Fragments		No. of Iterations	
	XFM-C	XFM	XFM-C	XFM	XFM-C	XFM	XFM-C	XFM	XFM-C	XFM
VS1	0	n/a	35	35	0	0	1.31%	1.31%	0	1
VS2	0	n/a	29	29	0	0	1.11%	1.11%	0	1
VS3	0	n/a	32	32	0	0	1.19%	1.19%	0	1
VS4	0	n/a	31	31	0	0	1.14%	1.14%	0	1
VS5	0	n/a	31	31	0	0	1.16%	1.16%	0	1
VS6	26	n/a	54	319	0	264	2.39%	14.13%	1	3
VS7	25	n/a	31	410	0	387	1.03%	13.61%	1	1
VS8	23	n/a	45	54	0	0	1.52%	1.82%	1	1
VS9	77	n/a	70	557	0	470	2.38%	18.96%	1	2
VS10	26	n/a	35	648	0	624	1.26%	23.28%	1	1
VS11	37	n/a	222	202	169	180	8.01%	7.28%	1	1
VS12	30	n/a	289	323	210	294	10.12%	11.31%	2	1
VS13	42	n/a	209	209	128	128	7.90%	7.90%	2	2
VS14	54	n/a	517	601	428	556	22.07%	25.65%	1	1
VS15	42	n/a	199	167	113	113	7.44%	6.25%	4	4

Table 8.4: Clusters and Candidates

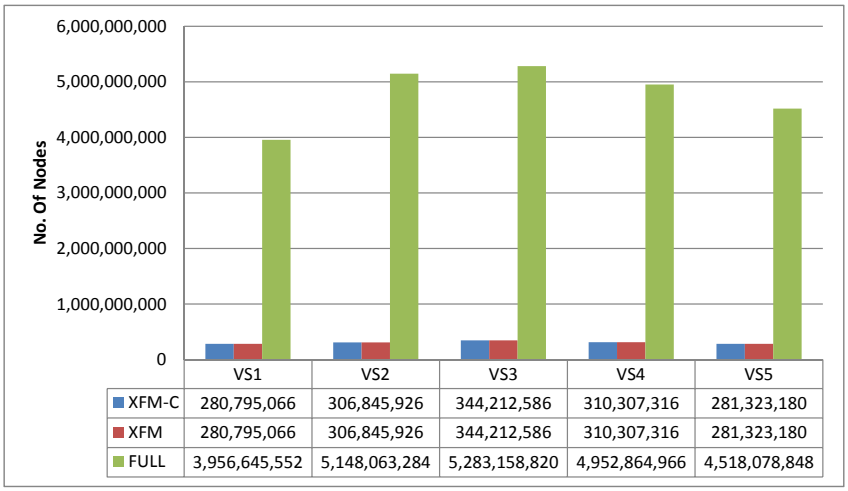
materialised fragments that are shared between views in cases where the FULL approach must access source data from the remote servers.

In §8.2.1 and §8.2.2, we record and analyse the following measures and costs, respectively:

1. Adaptation Cost: the total time required to achieve a sequence of changes.
2. No. of Candidates: the total number of fragments selected.
3. No. of VF fragments: the total number of VF fragments selected among the candidates.
4. Percentage of Instance Nodes: the percentage of instance nodes materialised by the XFM-C and XFM approaches relative to the FULL approach.
5. No. of Iterations: the number of iterations performed by the selection algorithm, where required.

8.2.1 Performance of Fragment Selection

As the fragment selection component is specific to the XFM view framework, we compare between our own approaches: the standard selection and selection with clustering. Those are the methods used in XFM and XFM-C, respectively. To compare with other approaches, we also show the costs when all views are materialised (FULL approach).



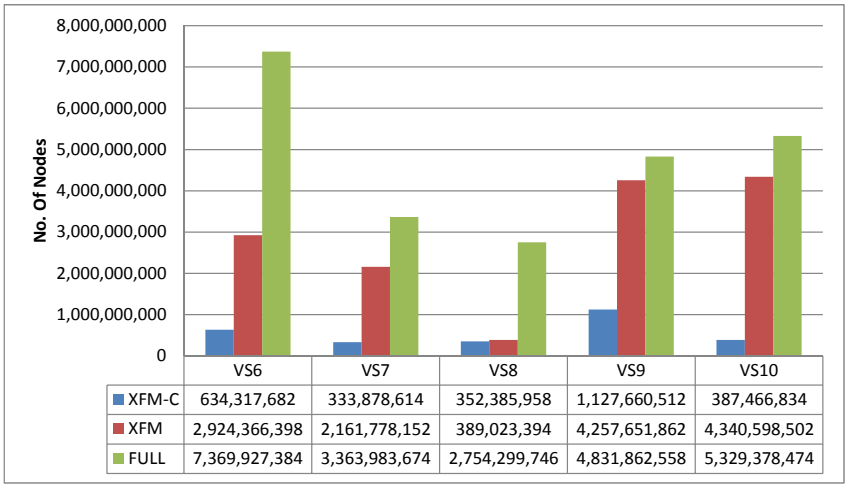
	VS1	VS2	VS3	VS4	VS5
XFM-C	280,795,066	306,845,926	344,212,586	310,307,316	281,323,180
XFM	280,795,066	306,845,926	344,212,586	310,307,316	281,323,180
FULL	3,956,645,552	5,148,063,284	5,283,158,820	4,952,864,966	4,518,078,848

Figure 8.1: Materialisation Cost for View Set VS1 - VS5

The purpose of fragment selection is to select the best fragments for materialisation. Should one materialise all views (VF fragments), this has the most positive impact on query performance as query processing is more efficient when all views are pre-computed and stored. Nevertheless, to fully materialise all views, will have a negative impact on view adaptation performance as every view change will impact on materialised views. Furthermore, full materialisation also increases view maintenance costs as large amount of nodes are stored and, additionally, many of them are duplicated. An optimised selection algorithm tries to balance the costs between query processing, view adaptation and view maintenance. In our approach, there are two key performance indicators for fragment selection: the cost of maintaining the materialised views and the number of VF fragments to be materialised. For the second indicator, we are seeking to materialise the smallest number of VF fragments. The fact that this is always the final fragment means that any change to the view definition must result in data adaptation.

For view set VS1 to VS5, since views are very similar and the overall costs of shared fragments are low, no clustering is required by the XFM-C approach. As a result, the selection is achieved by both XFM and XFM-C approaches within the first phase of the selection process, i.e., Fragment Selection phase. As shown in Figure 8.1, the materialisation cost of XFM and XFM-C are far less than the FULL approach, and as depicted in Table 8.4, they both select less than 1.5% of the overall set of fragments. The less nodes that have been materialised the less maintenance cost is required. In this case, no VF fragment is

selected which increases the chances that a change may not affect materialised fragments and thus, further improve the view adaptation performance. As will be shown in §8.2.2, the adaptation performance is much better when no View Fragment is materialised.

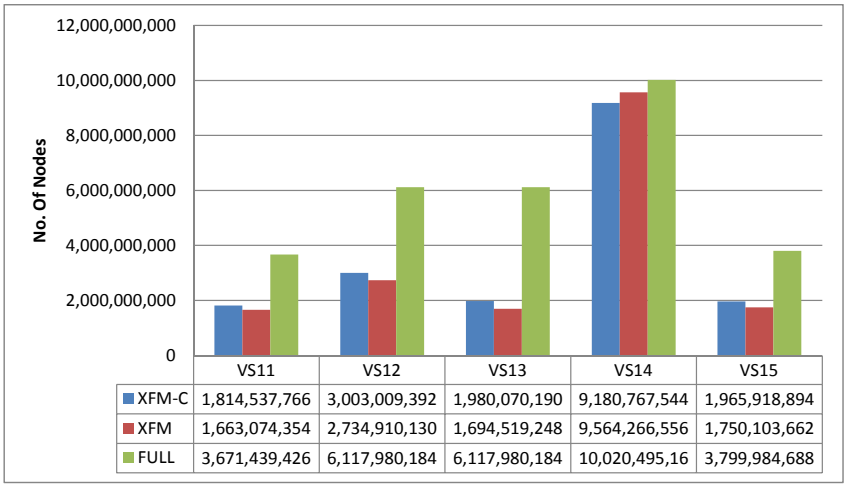


	VS6	VS7	VS8	VS9	VS10
XFM-C	634,317,682	333,878,614	352,385,958	1,127,660,512	387,466,834
XFM	2,924,366,398	2,161,778,152	389,023,394	4,257,651,862	4,340,598,502
FULL	7,369,927,384	3,363,983,674	2,754,299,746	4,831,862,558	5,329,378,474

Figure 8.2: Materialisation Cost for View Set VS6 - VS10

Views within the set VS6 and VS10 have low average similarity value and low overall costs for common fragments and as a consequence, views within those sets are clustered by the XFM approach, but no VF fragment is selected. As shown in Table 8.4, the XFM-C approach selects between 1.03% and 2.39% of the total fragments for materialisation, whereas the XFM approach selects more than 13.61% of the fragments with the exception of VS8. For VS8, although the XFM approach cannot complete the selection in the Fragment Selection phase, the sub-graph based selection in the View Coverage phase covers all views in one iteration as the remaining views are close to each other and have low overall costs. As shown in Figure 8.2, the XFM-C approach materialises far less instance nodes than the other two approaches as the consequence of applying the clustering technique. Additionally, as shown in Table 8.4, the XFM approach selects many VF fragments since the *Cost Evaluation* process is performed based on the context of the entire view graph and in such a case, costs do not reflect the real "importance" of each fragment to existing views. Nevertheless, the *Cost Evaluation* process of the XFM-C approach is performed based on each cluster to reflect the real significance of each fragment to existing views.

When views are either quite different or common fragments between views are costly, both XFM and XFM-C approaches select a large set of fragments for materialisation. As high-



	VS11	VS12	VS13	VS14	VS15
XFM-C	1,814,537,766	3,003,009,392	1,980,070,190	9,180,767,544	1,965,918,894
XFM	1,663,074,354	2,734,910,130	1,694,519,248	9,564,266,556	1,750,103,662
FULL	3,671,439,426	6,117,980,184	6,117,980,184	10,020,495,16	3,799,984,688

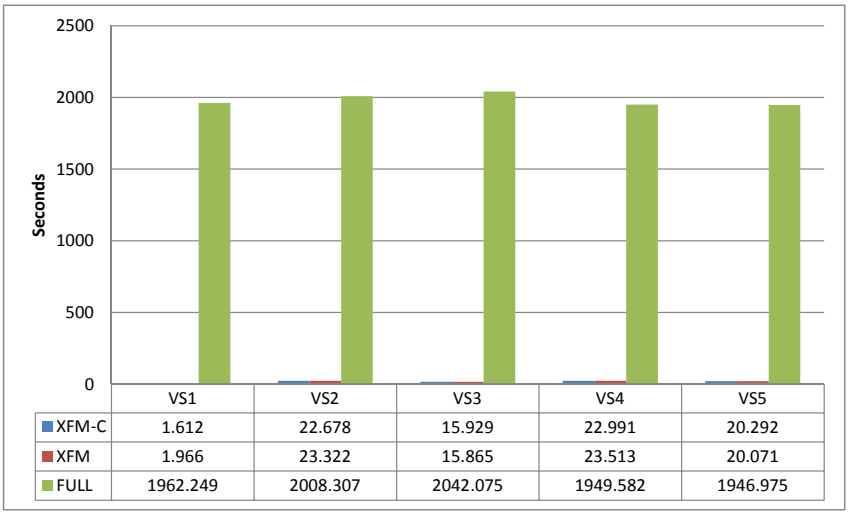
Figure 8.3: Materialisation Cost for View Set VS11 - VS15

lighted in Table 8.4, the number of fragments selected by both approaches vary, from 7% to 25% and large amounts of VF fragments are selected. As shown in Figure 8.3, in the worst case, i.e., VS14, three approaches materialise nearly the same amount of instance nodes and both XFM and XFM-C approaches require 4 iterations to cover all views. This type of case is unavoidable as when views are very different from each other or common fragments are too expensive to materialise, the only way to cover all views, for the purpose of query processing, is to materialise all VF fragments. Moreover, for the XFM-C approach, it materialises more nodes than the XFM approach for VS11, VS12, VS13 and VS15. This is due to the fact that a view may be covered by more than one fragment. For example, a view might be covered by candidates selected in the Fragment Selection phase, a fragment selected in the View Coverage phase by the clustering approach may also cover the same view, which increases the materialisation cost and also has more chances to influence the view adaptation performance.

In summary, the clustering-based approach requires much less materialisation costs than the other two approaches in terms of the number of fragments selected and the amount of nodes to materialise.

8.2.2 Performance of View Adaptation

The goal of this part of our evaluation is to demonstrate the improvements to view adaptation gained by the fragment-based approach, over the FULL approach. We do this by the



	VS1	VS2	VS3	VS4	VS5
XFM-C	1.612	22.678	15.929	22.991	20.292
XFM	1.966	23.322	15.865	23.513	20.071
FULL	1962.249	2008.307	2042.075	1949.582	1946.975

Figure 8.4: View Adaptation Cost for View Set VS1 - VS5

time required for a series of changes across the 15 view sets. Furthermore, we show our own adaptation performance under different selection plans.

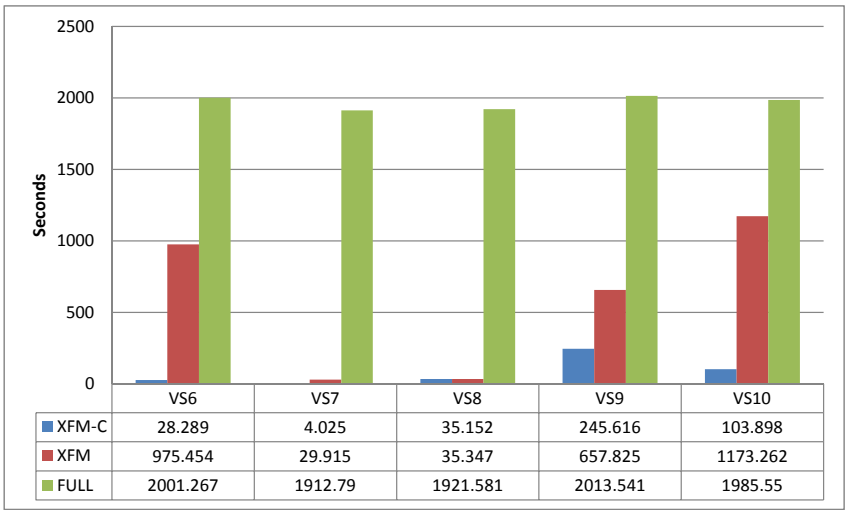
As part of the evaluation, there are two possibilities for data adaptation.

1. Recomputing by using an existing materialised fragment where the reused fragment could be the affected fragment itself or a shared materialised fragment depending on the type of change applied.

2. Recomputing by using data obtained from the database servers.

As shown in Figure 8.4, where no cluster was created (VS1-VS5), both XFM-C and XFM approaches have a similar adaptation performance. Furthermore, changes are handled far more efficiently by the XFM-C and XFM approaches than by the FULL approach. In this circumstance, views are relatively similar to each other and common fragments are relatively cheap for materialisation. Therefore, only a small number of fragments are materialised and changes applied to existing views may require only structural adaptation and no data is required to be adapted.

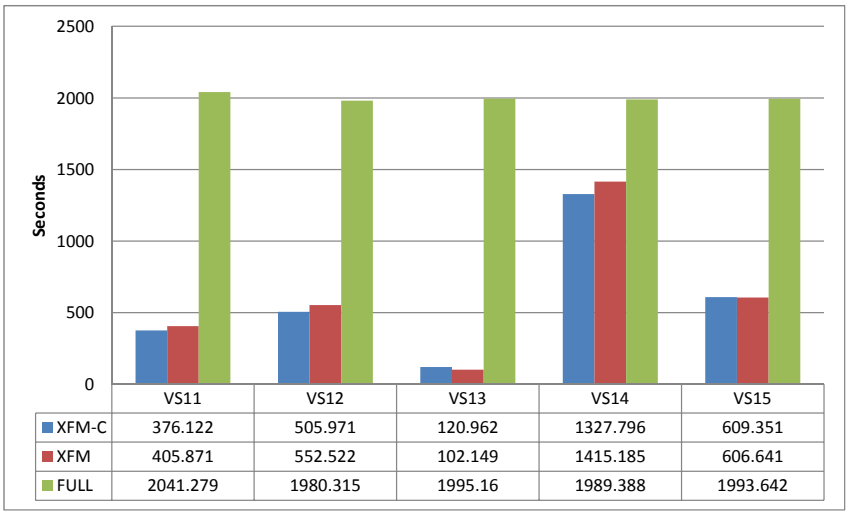
For VS6 to VS10, the XFM-C approach has far better adaptation performance than the other two approaches for set VS6, VS9 and VS10. This is because a large set of VF fragments are selected and materialised by the XFM approach, hence, when adaptation takes place, it must update those materialised VF fragments. Using the XFM-C superior fragment selection



	VS6	VS7	VS8	VS9	VS10
XFM-C	28.289	4.025	35.152	245.616	103.898
XFM	975.454	29.915	35.347	657.825	1173.262
FULL	2001.267	1912.79	1921.581	2013.541	1985.55

Figure 8.5: View Adaptation Cost for View Set VS6 - VS10

plan, the ability to manage changes with no data adaptation is improved and it has also a higher possibility to detect a fragment for reuse. For the cases that a view becomes more restricted after applying a change, the affected fragment is *self-maintainable* and the new sub-expression can be executed directly over the affected fragment provided that it is materialised.



	VS11	VS12	VS13	VS14	VS15
XFM-C	376.122	505.971	120.962	1327.796	609.351
XFM	405.871	552.522	102.149	1415.185	606.641
FULL	2041.279	1980.315	1995.16	1989.388	1993.642

Figure 8.6: View Adaptation Cost for View Set VS11 - VS15

Where existing views require a large number of VF fragments to be created, as in the case with VS11 to VS15, it may happen (e.g., VS14) that the XFM-C and XFM approaches obtain smaller performance gains against the FULL approach. This is due to the fact that a

large amount of VF fragments are selected and when changes make those VF fragments less restricted, both the XFM-C and XFM approaches must find a fragment for reuse where the affected fragments cannot be adapted from an existing fragment. However, for cases VS11, VS12, VS13 and VS15, most of the changes cause the fragment to be more restricted, which indicates that the affected fragments are self-maintainable and thus, the XFM-C and XFM approaches have far better adaptation performance.

8.3 Summary

In summary, both selection mechanisms perform well when compared to the FULL approach, both in terms of materialisation cost and view adaptation cost. The XFM-C approach delivers the best fragment selection and thus, has the biggest impact on both view adaptation performance and on materialisation cost. When views are very similar and the overall costs of common fragments are low, both XFM and XFM-C approaches make similar select decisions and have similar view adaptation performance, with both approaches far more efficient than the FULL approach. In our evaluation, we have a clear demonstration that the fragment based approach outperforms the FULL view approach.

When examining different approaches within our own system, our evaluation showed clear results. In the case where both the average similarity between views and the overall costs of common fragments are low, the XFM-C method makes better selection decision and obtains far better view adaptation performance. However, where views are very different and overall costs of common fragments are high, both XFM and XFM-C select a large set of VF fragments for materialisation and as a result, the view adaptation performance is not as good.

Chapter 9

Conclusions and Future Work

In this dissertation, we presented a fragment-based view adaptation approach for XPath views which are represented by XFM view graphs consisting of algebraic operators and fragments. The fragment-based view adaptation process operates on a global XFM view graph, where all views are integrated and when changes are applied, it adapts the graph both structurally, in terms of repositioning fragment nodes and operation nodes, and physically, where materialised fragments are updated. In this final chapter, we review the concepts presented and following that, we discuss areas where future research can be explored.

9.1 Thesis Summary

The initial hypothesis of this research was that view adaptation is more efficient when using a multi-view based framework with materialised data shared between views. Therefore, our research goals to deliver multi-view based approach can be reviewed.

1. It was necessary to define an XML view model with sufficiently expressive constructs and algebraic operators, to represent XML views.

2. We developed the view adaptation process which updates the view graph after applying changes to view definitions. This includes containment checking algorithm which identifies common expressions between views and determines the extent of changes between the old and new view definitions.

3. We developed a fragment selection process which decides the best fragments within

the view model for materialisation. This optimises the view adaptation process and balances the cost of query performance and view maintenance.

4. Finally, we devised a framework to enable each of the different components to interact.

A methodology was set to complete each goal in an incremental manner through the entire dissertation.

In Chapter 1, an introduction to XML databases and XML-based warehouse systems was presented and existing efforts of view adaptations were briefly discussed. It was concluded that, through lack of maturity, existing XML databases suffer from the performance issues. As a result, special attention was paid to view-based optimisation, which facilitates XML query processing.

In Chapter 2, a set of view adaptation mechanisms were discussed. Previous efforts focused mainly on view adaptation for relational views and among them, a fragment-based approach was demonstrated to be an optimised approach due to the fact that 1) materialised data is shared which avoids duplication; 2) the view adaptation performance is improved as there is a greater chance that materialised data can be reused; 3) the number of access operations to source data is decreased, which reduces the network communication cost and further improves view adaptation performance; and finally, 4) the fragment-based framework balances the query processing cost and view maintenance cost as only part of the fragments are materialised. However, due to different characteristics of relational data and XML data, the current fragment-based approach can only operate with relational views. Despite the fact that XML view adaptation approaches exist, there are several limitations, i.e., very limited changes are supported, no network communication cost is considered and, most importantly, views are treated as single entities, which faces the same problems as were encountered in relational world, e.g., duplication and decreased view adaptation performance.

To demonstrate how fragment-based adaptation is achieved in the XML world, we have proposed 1) a fragment-based view framework, where XPath views are merged and common parts are shared; 2) a containment checking algorithm to identify common expressions and the extent of changes between old view definitions and the new ones; 3) a cost-based

selection algorithm to select fragments for materialisation and a set of view adaptation algorithms to handle a classification of changes. Those four components were covered through Chapter 4 to Chapter 7 in this dissertation.

In Chapter 4, an introduction of the XFM view model and graph representation was presented. In our work, views are defined based on a subset of the XPath expressions, a core component of the XPath language. XPath Views are represented by XFM view graphs, which consist of algebraic operators and fragments. A transformation process is applied to transform XPath views first into the algebraic representation and then to the graph representation. We deliberately chose this path to allow further optimisation for our system, e.g., query rewriting. A merge algorithm was provided to integrate all view graphs into a *global* XFM view graph which serves as the basic framework for our view adaptation system.

In Chapter 5, a detailed description of our containment checking algorithm was given. In our approach, containment checking is achieved at the schema level with the assistance of a metadata construct, the SchemaGuide. We start by discussing relationships between XML trees, tree pattern queries, SchemaGuide and fragments. Based on those relationships, we proposed a containment checking algorithm to verify containment relationships between fragments within the global XFM view graph.

In Chapter 6, a cost-based fragment selection was presented. The selection plan is generated by applying a set of cost-based heuristics to each fragment and only those fragments that satisfy all heuristics are materialised. Based on whether all views are covered or not, further selection may be required. We demonstrated three selection algorithms, Graph-Based, Cluster-Based and Subgraph-Based methods, and the overall method uses a combination of those algorithms. As part of the selection process, Graph-Based selection uses costs estimated on the full view graph and applies cost-based heuristics to all fragments within the global view graph. It performs well when all views are relatively similar. The Cluster-Based and Subgraph-Based approaches refine the estimated costs in order to select any "missed" fragments within clusters and ensure all views are covered by selecting new fragments from a set of uncovered views (subgraphs).

In Chapter 7, a set of adaptation algorithms was presented which achieve view adaptation by means of a two-step process. The first step is to perform the structural adaptation, which updates the logical structure of the XFM view graph, and the second step is data adaptation

which updates the materialised fragments that are affected by the change. Our adaptation approach operates by first checking whether a change has an impact on views besides the target view. *FragmentReplication* is used to replicate fragments that are shared between target and non-target views and any fragment that is shared and affected by the change is then replicated. Based on the type of change, different adaptation algorithms are applied to maintain the logical structure of the graph and this is followed by the *FragmentOptimisation* which optimises the graph structure by exploring potential fragments to be shared. For data adaptation, we proposed five methods corresponding to different circumstances. From a high level perspective of view, the data adaptation process consists of two main methods: *incremental adaptation* and *recomputing*. Based on the two broad methods, we further refined them into five categories,

1. Self-Maintained Incremental Adaptation
2. Incremental Adaptation Using Local Fragments
3. Incremental Adaptation Using Remote Data
4. Recomputing Locally
5. Recomputing Remotely

They differ from each other by how views are adapted and XML instance nodes are obtained. Although recomputing may appear naive, in the context of XML databases, it is often the case that recomputing is faster than the incremental approach as updates are very inefficient for XML data.

We demonstrated in our evaluation chapter, the performance of view adaptation based on different selection strategies. We deliberately generated a large set of views to show the performance gain under different situations. In §8.2.1, we demonstrated the selection performance in terms of how selection plan affects the view adaptation, query processing and view maintenance. Materialising all views definitely has the best query processing performance, however, when it comes to view maintenance or view adaptation, it decreases view adaptation performance and increases maintenance cost. In §8.2.2, we demonstrated the performance of view adaptation comparing to the FULL approach. Our results demon-

strated that the view adaptation process has much better performance on a fragment-based framework.

9.2 Areas for Future Research

Based on what was learnt while delivering this research, we believe that there is a number of interesting research areas to be explored. We separate these into short term goals which can be achieved relatively quick and longer term goals, which require more prolonged research effort.

9.2.1 Short Term Research Goals

We think the following goals are relative easy to complete from a short term perspective and therefore, we set them as our primary research objects.

Clustering-Based Selection. One of the issues that the clustered-based approach faces is that a view is covered by more than one fragment. This is because in the Fragment Selection phase, the process checks only whether all views are covered or not, it ignores the actual fragments that have been selected in the Fragment Selection phase. As a result, when it reaches the clustering process in the View Coverage phase, the selection algorithm re-examines all fragments in each cluster and selects valid candidates. The fragments selected by the cluster-based method may cover same views as those fragments that were selected in the Fragment Selection phase. For some cases, this could be a valid action, i.e., when nodes contained by one fragment is a very small subset of the second fragment, then the first fragment is more "dedicated" to a small amount of views, whereas the second fragment is shared by a much larger set of views. Then it is necessary to materialise both of them to facilitate the adaptation process and also benefits the query processing in spite of that this is not our main concern here. When two fragments are shared by similar amount of views and contains similar amount of XML nodes, then it is not necessary to materialise both of them as this increases both maintenance and adaptation costs. Therefore, the first short term goal of our research is to refine our selection algorithm to analyse the difference between fragments and whether it is necessary to materialise multiple fragments to cover the same

set of views.

Incremental View Adaptation. We proposed different incremental methods to manage views that are either more or less restricted after applying a change. Incremental adaptation is a two-step process: 1) they first identify "redundant" or "extra" data depending on the type of changes applied; and 2) delete redundant data from the affected fragment or insert extra data into the affected fragment. Identifying data is relatively easy. However, deletion and insertion are more difficult as one must somehow maintain the original order and structural information between the XML nodes. As we have stated, a positional labelling/encoding scheme (index) can be used to maintain the order and structural information. The problem is how to efficiently perform the update process on it. We believe that the view adaptation process can provide further gains if it has access to the underlying database technology and exploit the XML index to develop update operators.

9.2.2 Longer Term Goals

The long term research goals of our research are to provide a more comprehensive framework to cover more XPath expressions, or XQuery expressions and to support more types of changes, e.g., grouping and aggregation.

The XFM Framework. Existing view-based approaches focus on a subset of XPath expressions, which is considered to be the core construct of the language. Most of the practical queries use this subset. However, we believe that from a long term perspective, as user requirements may vary, it is possible to extend our framework to support a larger set of XPath expressions or even XQuery expressions. This involves extending the XFM view graph with additional algebraic operators and fragment types and a more detailed containment checking algorithm based on the new view definition.

View Adaptation With Support of Grouping and Aggregation. As stated by [WLXB09], there is a compelling need of supporting analytical operations in XML queries, where grouping and aggregate functions are essential constructs of the queries. Another future goal for this research is to add support of grouping and aggregation to view adaptation

process.

Finally, we believe that as organisations continue to generate XML data through online services and transactions, the need to query and data mine these repositories for strategic information will grow. As a result, there is a significant impact that can be achieved through research that improves both functionality and speed of XML data.

Bibliography

[ABMP07] Andrei Arion, Véronique Benzaken, Ioana Manolescu, and Yannis Papakonstantinou. Structured Materialized Views for XML Queries. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 87–98. VLDB Endowment, 2007.

[AML+07] Padmapriya Ayyagari, Prasenjit Mitra, Dongwon Lee, Peng Liu, and Wang-Chien Lee. Incremental adaptation of XPath access control views. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, ASIACCS '07, pages 105–116, New York, NY, USA, 2007. ACM.

[AYCLS02] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Tree Pattern Query Minimization. *The VLDB Journal*, 11:315–331, December 2002.

[BaseX] BaseX. Visual Exploration and Querying of XML Data., 2010.

[Bel98] Zohra Bellahsene. View Adaptation in Data Warehousing Systems. In *Database and Expert Systems Applications, 9th International Conference, DEXA '98, Vienna, Austria, August 24-28, 1998, Proceedings*, volume 1460 of *Lecture Notes in Computer Science*, pages 300–309. Springer, 1998.

[Bel00] Zohra Bellahsene. Adapting Materialized Views after Redefinition in Distributed Environments. In *ER*, pages 239–252, 2000.

[Bel04] Zohra Bellahsene. View Adaptation in the Fragment-Based Approach. *IEEE Transactions Knowledge Data Engineering*, 16(11):1441–1455, 2004.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 310–321. ACM, 2002.

[BL03] Andreas Bauer and Wolfgang Lehner. On Solving the View Selection Problem in Distributed Data Warehouse Architectures. In *Proceedings of the 15th International Conference on Scientific and Statistical Database Management*, SSDBM '03, pages 43–54, Washington, DC, USA, 2003. IEEE Computer Society.

[BOB+04] Andrey Balmin, Fatma Özcan, Kevin S. Beyer, Roberta J. Cochrane, and Hamid Pirahesh. A Framework for Using Materialized XPath Views in XML Query Processing. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, volume 30 of *VLDB '04*, pages 60–71. VLDB Endowment, 2004.

[Bra03] Chris Brandin. XML Data Management: Information Modeling with XML, May 2003.

[CBHB09] Leonardo Weiss F. Chaves, Erik Buchmann, Fabian Hueske, and Klemens Böhm. Towards Materialized View Selection for Distributed Databases. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 1088–1099, New York, NY, USA, 2009. ACM.

[CHS02] Rada Chirkova, Alon Y. Halevy, and Dan Suciu. A Formal Perspective on the View Selection Problem. *The VLDB Journal*, 11:216–237, November 2002.

[eXist] eXist. eXist-db Open Source Native XML Database, 2000.

[FLZ07] Jian-Hua Feng, Yu-Guo Liao, and Yong Zhang. HCH for Checking Containment of XPath Fragment. *Journal of Computter Science and Technology*, 22:736–748, September 2007.

[FTU98] Carles Farré, Ernest Teniente, and Toni Urpí. Query Containment Checking as a View Updating Problem. In *Proceedings of the 9th International Conference on Database and Expert Systems Applications*, DEXA '98, pages 310–321, London, UK, 1998. Springer-Verlag.

[FTU99] Carles Farré, Ernest Teniente, and Toni Urpí. The Constructive Method for Query Containment Checking. In *Proceedings of the 10th International Conference on Database and Expert Systems Applications*, DEXA '99, pages 583–593, London, UK, 1999. Springer-Verlag.

[GCV09] Haris Georgiadis, Minas Charalambides, and Vasilis Vassalos. Cost Based Plan Selection for XPath. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 603–614, New York, NY, USA, 2009. ACM.

[GMR95] Ashish Gupta, Inderpal Singh Mumick, and Kenneth A. Ross. Adapting Materialized Views after Redefinitions. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 211–222. ACM Press, 1995.

[GMRR01] Ashish Gupta, Inderpal S. Mumick, Jun Rao, and Kenneth A. Ross. Adapting Materialized Views after Redefinitions: Techniques and a Performance Study. *Information Systems*, 26:323–362, July 2001.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. *SIGMOD Rec.*, 22:157–166, June 1993.

[GPSH02] Antara Ghosh, Jignashu Parikh, Vibhuti S. Sengar, and Jayant R. Haritsa. Plan Selection Based on Query Clustering. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 179–190. VLDB Endowment, 2002.

[Gru02] Torsten Grust. Accelerating XPath Location Steps. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 109–120, New York, NY, USA, 2002. ACM.

[GRV01] Matteo Golfarelli, Stefano Rizzi, and Boris Vrdoljak. Data Warehouse Design from XML Sources. In *Proceedings of the 4th ACM International Workshop on Data Warehousing and OLAP*, DOLAP '01, pages 40–47, New York, NY, USA, 2001. ACM.

[GW97] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 436–445, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[GZ08] An Gong and Weijing Zhao. Clustering-Based Dynamic Materialized View Selection Algorithm. In *Fifth International Conference on Fuzzy Systems and Knowledge Discovery, 2008. FSKD '08.*, volume 5, pages 391 –395, October 2008.

[IHH09] Sayyed Kamyar Izadi, Theo Härder, and Mostafa S. Haghjoo. S3: Evaluation of Tree-Pattern XML Queries Supported by Structural Summaries. *Data & Knowledge Engineering*, 68:126–145, January 2009.

[JL10] Xueyun Jin and Husheng Liao. An algorithm for Incremental Maintenance of Materialized XPath View. In *Proceedings of the 11th International Conference on Web-Age Information Management*, WAIM'10, pages 513–524, Berlin, Heidelberg, 2010. Springer-Verlag.

[JW02] Glen Jeh and Jennifer Widom. SimRank: A Measure of Structural-Context Similarity. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 538–543, New York, NY, USA, 2002. ACM.

[KGJ10] T.V. Vijay Kumar, Anurag Goel, and Neeraj Jain. Mining Information for Constructing Materialised views. *Int. J. Inf. Commun. Techol.*, 2:386–405, August 2010.

[LLLL04] Bo Luo, Dongwon Lee, Wang-Chien Lee, and Peng Liu. QFilter: fine-grained run-time XML access control via NFA-based query rewriting. In *Proceedings*

of the thirteenth ACM international conference on Information and knowledge management, CIKM '04, pages 543–552, New York, NY, USA, 2004. ACM.

[LLLL05] Bo Luo, Dongwon Lee, Wang-Chien Lee, and Peng Liu. Deep Set Operators for XQuery. In Daniela Florescu and Hamid Pirahesh, editors, *Proceedings of the Second International Workshop on XQuery Implementation, Experience and Perspectives <XIME-P/>, in cooperation with ACM SIGMOD, June 16-17, 2005, Baltimore, Maryland, USA*, 2005.

[LR10] Jun Liu and Mark Roantree. OTwig: An Optimised Twig Pattern Matching Approach for XML Databases. In *SOFSEM 2010: Theory and Practice of Computer Science, 36th Conference on Current Trends in Theory and Practice of Computer Science, Spindleruv Mlýn, Czech Republic, January 23-29, 2010. Proceedings*, volume 5901 of *Lecture Notes in Computer Science*, pages 564–575. Springer, 2010.

[LR11] Jun Liu and Mark Roantree. Selecting Fragments in a Partially Materialized XML View Graph. Paper in Submission to TKDE, May 2011.

[LRB10a] Jun Liu, Mark Roantree, and Zohra Bellahsene. A SchemaGuide for Accelerating the View Adaptation Process. In *29th Interational Conference on Conceptual Modeling (ER'10)*, volume 6412, pages 160–173. Springer, 2010.

[LRB10b] Jun Liu, Mark Roantree, and Zohra Bellahsene. Optimizing XML Data with View Fragments. In Shen H.T. and A. Bouguettaya, editors, *21st Australasian Database Conference (ADC 2010)*, volume 104 of *CRPIT*, pages 151–160, Brisbane, Australia, 2010. ACS.

[LRB11] Jun Liu, Mark Roantree, and Zohra Bellahsene. A Fragment-Based View Adaptation Mechanism for XPath Views. Paper in Submission to DKE, Feb 2011.

[LSK07] Ki Yong Lee, Jin Hyun Son, and Myoung Ho Kim. Reducing the Cost of Accessing Relations in Incremental View Maintenance. *Decis. Support Syst.*, 43:512–526, March 2007.

[LWC06] Eric Jui-Lin Lu, Bo-Chan Wu, and Po-Yun Chuang. An Empirical Study of XML Data Management in Business Information Systems. *Journal of Systems and Software*, 79(7):984 – 1000, 2006.

[LWH10] Ki-Hoon Lee, Kyu-Young Whang, and Wook-Shin Han. XMin: Minimizing Tree Pattern Queries with Minimality Guarantee. *World Wide Web*, 13:343–371, 2010. 10.1007/s11280-010-0089-x.

[LWZ06] Laks V. S. Lakshmanan, Hui Wang, and Zheng Zhao. Answering Tree Pattern Queries Using Views. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 571–582. VLDB Endowment, 2006.

[MD96] Mukesh K. Mohania and Guozhu Dong. Algorithms for Adapting Materialised Views in Data Warehouses. In *CODAS*, pages 309–316, 1996.

[MD09] Hadj Mahboubi and Jérôme Darmont. Enhancing XML Data Warehouse Query Performance by Fragmentation. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1555–1562, New York, NY, USA, 2009. ACM.

[Moh97] Mukesh Mohania. Avoiding Re-computation: View Adaptation in Data Warehouses. In *In Proc. of 8 th International Database Workshop, Hong Kong*, pages 151–165, 1997.

[MonetDB] MonetDB. MonetDB/XQuery, 2008.

[MRMW09] Dónall McCann, Mark Roantree, Niall Moyna, and Michael Whelan. Synchronizing Sensed Data in Team Sports. *ERCIM News*, 2009(76), 2009.

[MRS11] Gerard Marks, Mark Roantree, and Dominick Smyth. Optimising Queries for Web Generated Sensor Data. In *In The 22nd Australasian Database Conference, Perth, Australia, 2011*, 2011.

[MS02] Gerome Miklau and Dan Suciu. Containment and Equivalence for an XPath Fragment. In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '02)*, pages 65–76, New York, NY, USA, 2002. ACM.

[MS04] Gerome Miklau and Dan Suciu. Containment and Equivalence for a Fragment of XPath. *Journal of the ACM*, 51:2–45, January 2004.

[NDRR06] Vicky Nassis, Tharam Dillon, Rajugan Rajagopalapillai, and Wenny Rahayu. An XML Document Warehouse Model. In Mong Li Lee, Kian-Lee Tan, and Vilas Wuwongse, editors, *Database Systems for Advanced Applications*, volume 3882 of *Lecture Notes in Computer Science*, pages 513–529. Springer Berlin / Heidelberg, 2006.

[Nev02] Frank Neven. Automata Theory for XML Researchers. *SIGMOD Rec.*, 31:39–46, September 2002.

[NNNT02] Tapio Niemi, Marko Niinimäki, Jyrki Nummenmaa, and Peter Thanisch. Constructing an OLAP cube from distributed XML data. In *Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP*, DOLAP '02, pages 22–27, New York, NY, USA, 2002. ACM.

[NS03] Frank Neven and Thomas Schwentick. XPath Containment in the Presence of Disjunction, DTDs, and Variables. In *Proceedings of the 9th International Conference on Database Theory*, ICDT '03, pages 315–329, London, UK, 2003. Springer-Verlag.

[OCMH05] Makoto Onizuka, Fong Yee Chan, Ryusuke Michigami, and Takashi Honishi. Incremental Maintenance for Materialized XPath/XSLT Views. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 671–681, New York, NY, USA, 2005. ACM.

[OMFB02] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking Forward. In Akmal B. Chaudhri, Rainer Unland, Chabane Djeraba, and Wolfgang Lindner, editors, *XML-Based Data Management and Multimedia Engineering - EDBT 2002 Workshops, EDBT 2002 Workshops XMLDM, MDDE, and YRWS, Prague, Czech Republic, March 24-28, 2002, Revised Papers*, volume 2490 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 2002.

[OR10] Martin F. O'Connor and Mark Roantree. Desirable Properties for XML Update Mechanisms. In *Proceedings of the 2010 EDBT/ICDT Workshops*, EDBT '10, pages 23:1–23:9, New York, NY, USA, 2010. ACM.

[PP03] Dennis Pedersen and Torben Bach Pedersen. Achieving Adaptivity for OLAP-XML Federations. In *Proceedings of the 6th ACM international workshop on Data warehousing and OLAP*, DOLAP '03, pages 25–32, New York, NY, USA, 2003. ACM.

[PPP04] D. Pedersen, J. Pedersen, and T.B. Pedersen. Integrating XML Data in the TARGIT OLAP System. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 778 – 781, Apr 2004.

[PRP02] D. Pedersen, K. Riis, and T.B. Pedersen. XML-extended OLAP querying. In *Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on*, pages 195 – 206, 2002.

[QYD07] Lu Qin, Jeffrey Xu Yu, and Bolin Ding. Twiglist: Make Twig Pattern Matching Fast. In *Proceedings of the 12th international conference on Database systems for advanced applications*, DASFAA'07, pages 850–862, Berlin, Heidelberg, 2007. Springer-Verlag.

[RRT04] Laura Rusu, Wenny Rahayu, and David Taniar. On Building XML Data Warehouses. In Zheng Yang, Hujun Yin, and Richard Everson, editors, *Intelligent Data Engineering and Automated Learning ?IDEAL 2004*, volume 3177 of *Lecture Notes in Computer Science*, pages 293–299. Springer Berlin / Heidelberg, 2004.

[RS09] Mark Roantree and Mikko Sallinen. Introduction - The Sensor Web - Bridging the Physical-Digital Divide. *ERCIM News*, 2009(76), 2009.

[RSSB00] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. *SIGMOD Rec.*, 29:249–260, May 2000.

[RTTZ10] Franck Ravat, Olivier Teste, Ronan Tournier, and Gilles Zurfluh. Finding an Application-Appropriate Model for XML Data Warehouses. *Information Systems*, 35:662–687, September 2010.

[Sal05] Airi Salminen. Building Digital Government by XML. In *Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 5 - Volume 05*, pages 122.2–, Washington, DC, USA, 2005. IEEE Computer Society.

[Sch04] Thomas Schwentick. XPath Query Containment. *SIGMOD Rec.*, 33:101–109, March 2004.

[STP+05] Arsany Sawires, Junichi Tatemura, Oliver Po, Divyakant Agrawal, and K. SelÇuk Candan. Incremental Maintenance of Path-Expression Views. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 443–454, New York, NY, USA, 2005. ACM.

[TW05] TPC-W. TPC-W: Transactional Web e-Commerce Benchmark, April 2005.

[TYÖ+08a] Nan Tang, Jeffrey Xu Yu, M. Tamer Özsu, Byron Choi, and Kam-Fai Wong. Multiple Materialized View Selection for XPath Query Rewriting. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 873–882. IEEE, 2008.

[TYÖ+08b] Nan Tang, Jeffrey Xu Yu, M. Tamer Özsu, Byron Choi, and Kam-Fai Wong. Multiple Materialized View Selection for XPath Query Rewriting. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 873–882. IEEE, 2008.

[W3C04] W3C. XML Schema, Oct 2004.

[W3C08] W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition), November 2008.

[W3C10a] W3C. XML Path Language (XPath) 2.0 (Second Edition), December 2010.

[W3C10b] W3C. XQuery 1.0: An XML Query Language (Second Edition), December 2010.

[W3C10c] W3C. XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition), December 2010.

[W3C10d] W3C. XSLT 2.0 and XQuery 1.0 Serialization (Second Edition) - http://www.w3.org/TR/xslt-xquery-serialization/, December 2010.

[W3C11] W3C. XQuery Update Facility 1.0, March 2011.

[WLXB09] Huayu Wu, Tok Wang Ling, Liang Xu, and Zhifeng Bao. Performing grouping and aggregate functions in XML queries. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 1001–1010, New York, NY, USA, 2009. ACM.

[WLY11] Junhu Wang, Jiang Li, and Jeffrey Xu Yu. Answering Tree Pattern Queries Using Views: A Revisit. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 153–164, New York, NY, USA, 2011. ACM.

[Woo01] Peter T. Wood. Minimising Simple XPath Expressions. In *WebDB*, pages 13–18, 2001.

[Woo03] Peter T. Wood. Containment for XPath Fragments under DTD Constraints. In *Proceedings of the 9th International Conference on Database Theory*, ICDT '03, pages 300–314, London, UK, 2003. Springer-Verlag.

[WTW09] Xiaoying Wu, Dimitri Theodoratos, and Wendy Hui Wang. Answering XML Queries Using Materialized Views Revisited. In *Proceeding of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 475–484, New York, NY, USA, 2009. ACM.

[Xerces2] Xerces2 Parser. The Apache Xerces Project - http://xerces.apache.org/.

[YGYL05] Wei Ye, Ning Gu, Genxing Yang, and Zhenyu Liu. Extended Derivation Cube Based View Materialization Selection in Distributed Data Warehouse. In Wen-

fei Fan, Zhaohui Wu, and Jun Yang, editors, *Advances in Web-Age Information Management*, volume 3739 of *Lecture Notes in Computer Science*, pages 245–256. Springer Berlin / Heidelberg, 2005.

[ZLBT03] Ji Zhang, Tok Ling, Robert Bruckner, and A Tjoa. Building XML Data Warehouse Based on Frequent Patterns in User Queries. In Yahiko Kambayashi, Mukesh Mohania, and Wolfram W?, editors, *Data Warehousing and Knowledge Discovery*, volume 2737 of *Lecture Notes in Computer Science*, pages 99–108. Springer Berlin / Heidelberg, 2003.

[ZLE07] Jingren Zhou, Per-Ake Larson, and Hicham G. Elmongui. Lazy Maintenance of Materialized Views. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 231–242. VLDB Endowment, 2007.

[ZWLZ05] Ji Zhang, Wei Wang, Han Liu, and Sheng Zhang. X-Warehouse: Building Query Pattern-Driven Data Warehouse For XML Data. In *Special interest tracks and posters of the 14th international conference on World Wide Web*, WWW '05, pages 896–897, New York, NY, USA, 2005. ACM.