

Chapter 1

Introduction

Extensible Markup Language (XML) has emerged as a standard for representing and exchanging information on the Web. XML is widely accepted for its several important capabilities. Among them are *extensibility* (XML content can be easily extended by both technical and non-technical user-groups), *flexibility* (users can create their own tags to support meaning of specific content) and *interoperability* (many domain applications have adopted XML as a data interchange standard).

The adoption of XML is increasing rapidly. However, despite its popularity in both research work and industrial applications, XML data management still faces many challenges such as semantic specification, XML query processing and manipulation (e.g. XPath, XQuery, XSL, etc), and query optimization, to name a few.

In XML data management models, an XML schema¹ is normally used to structure and constrain XML content. Among existing XML-based schemas, *XML Schema*² is rich in semantics and has become a W3C recommendation [W3C, 2004a; 2004b]. The new features supported by W3C are available only in XML Schema.

Most XML documents are developed with an underlying XML Schema [Li & Miller, 2005]. Semantics in *XML Schema* are very useful for many XML-related areas such as query satisfaction optimization and many others. The semantics in XML Schema

¹ Refer to XML schema in general.

² Refer to XML schema in XSD format.

which is used for transforming XML queries to equivalent queries has attracted renewed interest from the research community.

This thesis investigates semantic transformation methodologies using schema knowledge to transform an XML query to its equivalent semantic query. The query performance (accuracy and execution speed) is then evaluated, which allows us to identify semantic transformation typologies used as generalized optimization devices.

1.1 XML: Background

In 1996, the W3C announced the development of XML, which was based on SGML, a Standard Generalized Markup Language defined by ISO [ISO, 1986]. The consortium acknowledged that HTML [Conklin, 1987] - a subset of SGML, could not function as a meta-language for describing mark-up languages, for it has rigid tags that were designed for web presentation purposes.

XML was first endorsed by W3C in 1998 as a meta-language suitable for describing mark-up languages [W3C, 1998]. XML tags are flexible and data relationships can exist between tags, making XML self-documenting.

```
<?xml version="1.0" encoding="UTF-8"?>
<PART>
  <TITLE>Computer Parts</TITLE>
  <CONFIGURATION>
    <ITEM>Motherboard</ITEM>
    <MANUFACTURER>ASUS</MANUFACTURER>
    <MODEL>P3B-F</MODEL>
    <REMARK> Original</REMARK>
  </CONFIGURATION>
</PART>
```

Figure 1.1 Example of XML Fragment

Its rich semantic extensibility and flexibility have made XML an open standard for storing and exchanging information among applications and on the World Wide Web. An example of an XML fragment is shown in Figure 1.1 which models a computer PART. It is a well-formed, self-described document showing PART with

other entities such as TITLE and CONFIGURATION. The entity CONFIGURATION has its own entities such as ITEM, MANUFACTURER, MODEL and REMARK. Each entity is enclosed by a pair of meaningful tags.

The simplicity of XML has enabled the Web to provide more effective publishing and information exchange facilities. The online exchange of information and requirements is no longer an issue. However, due to its convenience, the amount of XML-facilitated information continues to increase rapidly. Existing challenges such as performance, integrity and efficient storage are some of the many problems that need to be addressed.

1.2 XML Databases

With the ubiquitous use of XML, efficient storage of XML data is becoming a critical concern. XML document storage needs to be efficient; in addition, the management of persistent XML data requires the capabilities to deal with data independence, integration, access rights, versions, views, integrity, redundancy, consistency, recovery, and the enforcement of standards [Salminen and Tompa 2001].

Any repository that can store XML data is categorized as an XML database. There are two main XML database categories including Native XML database and XML-Enabled database. Any database engine that has originally been constructed and developed to manage XML data is known as a Native XML Database. XML-enabled databases are those that are extended with XML capability from any existing relational engine. Hence, any Relational Database Management System (RDBMS) that extends its functionality to include XML data management capabilities is known as an XML-enabled database system [Bourret, 2005].

While Native XML database systems preserve XML document structures and store these natively, most XML-enabled database systems shred and store XML data in relational or object-relational structures. Very few XML-enabled database systems can store XML data natively; such XML-enabled database systems are referred to as Hybrid XML databases [Beyer et al., 2006]. Since the introduction of the Hybrid

XML database, it has often been referred to as a Hybrid XML Relational database [Baqasah & Pardede, 2010; Jensen, et al., 2006; Moro, et al., 2007]. In fact, any database (e.g. relational, object-oriented, etc.) is extended to support XML database, and storing XML data natively is considered as a Hybrid XML database. Hence, an important advantage of a Hybrid XML database is that it can facilitate complete interoperability of XML and other databases (e.g. Relational or Object-Oriented) storage paradigms [Beyer et al., 2006 and Stromback et al., 2009].

XML documents stored in an XML database can be divided into two categories: document-centric and data-centric [Bourret, 2005]. The term document-centric refers to XML documents in which information is expressed with no standard structure required. Such XML documents include user manuals, static web pages, or marketing leaflets. Native XML databases are most commonly used to store document-centric documents as they preserve features such as full-text searches of certain portions of documents [Bourret, 2005]. The term ‘data-centric’ refers to XML documents in which information is expressed with a standard required structure such as scientific data, or customer details: names and addresses, etc. An XML-enabled database is grouped under the data-centric category due to its regular record structure.

The structural storage requirement of XML data can become very complex in XML-enabled databases. Several storage techniques for XML have been implemented by different database vendors such as Oracle [Oracle, 2010], DB2 pureXML [IBM, 2009]. A simple solution is to store XML data as Character Large Object Binary (CLOB); this technique is good for retrieving and inserting whole documents but not for data processing as query performance faces significant problems. A slightly more advanced storage technique is to shred XML into relational tables or object-relational tables, which enables better query processing and performance than does the CLOB storage technique. However, storing XML data in relational or object-relational tables requires flattening the hierarchical structure of XML documents (i.e. relational tables are used). A more advanced storage technique is to use an object-relational method of storing XML data, that is, using an XML Schema for object mapping and creating relations. The most advanced technique is to store XML data natively (e.g.

binary technique with XML Schema validation). Nevertheless, each technique poses its own challenges.

XML-enabled databases and Hybrid XML Relational databases are based on existing RDBMS products which have been augmented to provide XML support. For example, prominent vendors such as IBM and Oracle have developed their respective products to support XML. These vendors recognize the growing popularity of, and support for, XML data structurally, and understand that without XML support, their RDBMS would soon start to lose market share [Malloy & Mlynkova, 2009].

1.3 XML Schema & Its Importance

Document Type Definition (DTD) is a recommended schema by W3C which has been released to facilitate the structure of XML documents. However, many problems have been identified by the user community which have prompted the W3C to look into the development of XML Schema. Problems faced by DTD include: limited support for data types, no namespace awareness, and a structuring element is nested only within other elements.

To address the shortcomings of DTD, many schema languages have been proposed. These include, to name just a few: XML Schema [Sperberg-McQueen and Thompson, 2005], Document Structure Description (DSD) [Klarlund et al., 2002], Relax NG [Clark and Murata 2001] and Schematron [Jelliffe, 2005]. Of these, XML Schema has been recommended by W3C.

The role of XML Schema is to constrain and structure XML content. Certain data types for defining data elements may enable storage space efficiency. For example, an *integer* data type used to declare a particular data element in the document would create better memory space than would a *string* data type.

XML Schema plays several important roles in storage management including validation of the XML documents. Documents that are declared with a particular schema will need to use the schema to validate the documents during the insertion,

deletion or updating of data into or from the databases. Some database systems require schema validation prior to undertaking any data manipulation.

Furthermore, information in XML Schema plays a significant role in query optimization. The creation of internal objects may influence the way in which a query is processed. Some XML databases allow the shredding of XML documents and store these in a set of tables. The set of tables and internal relationships among them are automatically derived from the XML Schema. A well-designed XML Schema facilitates better underlying structures thereby allowing queries to be processed more efficiently.

1.4 XML: Query Languages

Since the arrival of XML, several XML query languages have emerged for the manipulation of data. The query languages include Lorel [Abiteboul et al., 1997], XPath 1.0 [W3C, 1999], XPath 2.0 [W3C, 1999; 2007a; 2010], XML-QL [Deutsch et al., 1999], XML-GL [Ceri et al., 1999], and XQuery [W3C, 2007b]. XPath and XQuery are still currently being used and have attracted a number of research works on performance- related issues.

XPath is a navigational query language that is designed to access parts of XML documents. It can support single or branching path structures by means of predicates for filtering content. For navigating XML documents, XPath has a full set of axes including *child*, *descendant*, *descendant-or-self*, *parent*, *ancestor*, *ancestor-or-self*, *preceding*, *preceding-sibling*, *following*, *following-sibling*, *self*, *attribute* and *namespace*. XQuery is derived from an XML query language named Quilt [Chamberlin et al., 2000]. It borrows some features from other languages including XPath. XQuery operates on the abstract and logical structure of an XML document [W3C, 2007b].

Due to the ever-increasing adoption of XML, there is a need to ensure that XML query languages perform efficiently. Query optimization and transformation for XML query languages, both syntactically and semantically, have received much attention from research communities in recent years. However, due to the rapid

development of XML data management, query optimization still requires much attention.

Semantic query optimization utilizes schema constraints to directly optimize a given query with a set of optimization rules. Due to the current complexity of the XML data structure, which is enabled by rich semantics in XML Schema, semantic query optimization should be done in a more systematic manner. For a complete solution, we leverage the semantics from XML Schema and use it to investigate a set of semantic transformation typologies. The semantic transformation typologies will be empirically tested for performance evaluation by using a large set of XPath queries. The performance results will be analyzed in order to identify semantic transformation typologies as optimization devices.

1.5 Motivation

For any data management model and query language, the ability to rewrite queries into equivalent queries is needed for several data management purposes. Of these, rewriting a query with the purpose of query optimization plays an important role; this is the case when rewritten queries can be evaluated more efficiently. Of the existing query rewriting techniques, semantic query transformation is the one that uses only semantics in the schema to transform any predefined user query to improve performance.

In the early 1980s, semantic query transformation [Chakravarthy et al., 1990; Hammer and Jondik, 1980; King, 1981a and 1981b; Shenoy and Ozsoyoglu, 1987] was introduced to utilize knowledge in the schemas to enable query rewriting. The principle of semantic query transformation is to use certain transformation rules that are developed with knowledge within the schema to reformulate a given query into its equivalent query. Semantically equivalent query is expected to perform better than its original query.

As the process of applying semantics to optimize queries has been adopted by other databases including Relational, Object-Oriented and Deductive databases, it is also useful in XML databases due to the availability of XML Schemas. However, there is

a difference in terms of constraints among the database types. XML constraints in XML Schema are driven by two sets of constraints: structural constraints (content model) and constraints of elements. Semantic query transformation for XML queries requires multiple semantics in most cases.

For example, in a collection of theses, is it possible to find theses that must have at least one supervisor who is referenced to an existing thesis? With such rules, we can determine from the schema that in order for a research student to be entered into the database, the student must have a valid supervisor who is referenced by an existing PhD thesis. These rules can be formulated to eliminate the unnecessary execution of a condition in the query if the structural, referenced and cardinality constraints of query components are correctly described. The benefit of having this rule is that the query components can be reduced prior to the processing stage and which in turn results in better performance.

The challenge of semantic transformation is the processing of schema information. Processing techniques must be carefully designed and efficient enough that overheads incurred by searching for the semantics can be kept to a minimum.

In addition to the above, semantic query transformation has the benefit of increasing performance significantly, if constraint or semantic conflicts in the query can be detected prior to reaching the database execution level.

For example, consider queries that contradict the semantics defined in the schema: without semantic transformation, how long would the conflict query take to process before the user is notified that the requested information is not available in the database? Semantic query transformation takes advantage of constraints defined in the XML Schema to transform the query. A query can be rejected if the semantics used by the query cannot be mapped to those available in the schemas. The performance comparison is based on the transformation time of the query and the query that executes against the database. This is because the conflict query can never reach the execution stage.

1.6 Objectives

XML Schema offers rich semantic capabilities providing both structural and data semantics for validation of XML documents or data. While it has continued to gain support from W3C, DTD has become obsolete and as a result of its lack of semantics it cannot be used to accomplish many XML-related tasks. Yet semantic query optimization still utilizes DTD to achieve query optimization tasks. Nevertheless, this will soon change, as the most recent XML documents have been developed using XML Schema, enabling the use of its semantics for semantic query transformation as part of the optimization research area.

The main intention of this thesis is to utilize the semantic capabilities of the XML Schema, and to apply them to the development of semantic transformation typologies to transform XPath queries to equivalent but semantically restructured XPath queries. The semantic transformation typologies are implemented and their performance is evaluated. The results are used to identify semantic transformations that can become generalized optimization devices.

The objectives of this thesis can be summarized as follows:

- To provide a suitable semantic transformation methodology to transform XPath queries to equivalent but semantically restructured XPath queries by exploiting semantics defined in the XML Schema.
- To ensure that the transformed semantic XPath queries and original XPath queries produce the same result, i.e. establishing *equivalence*.
- To ensure that XML Schema semantics are well-derived in order to support semantic transformation of XPath queries.
- To ensure that semantic transformations are efficiently evaluated, in order to identify individual semantic transformation as an optimization device (*technique*).

1.7 Scope and Plan of the Thesis

A *scope* and a *plan* require a *context*. We delineate some fundamental requirements that are essential to support our proposed methodology.

- a. XML Schema provides essential resources which has made our research possible.
The minimal requirement for this research is the XML Schema.

In choosing a method of XML storage management, we consider any XML database repository that is equipped with a facility to validate XML documents against XML Schemas in this research.

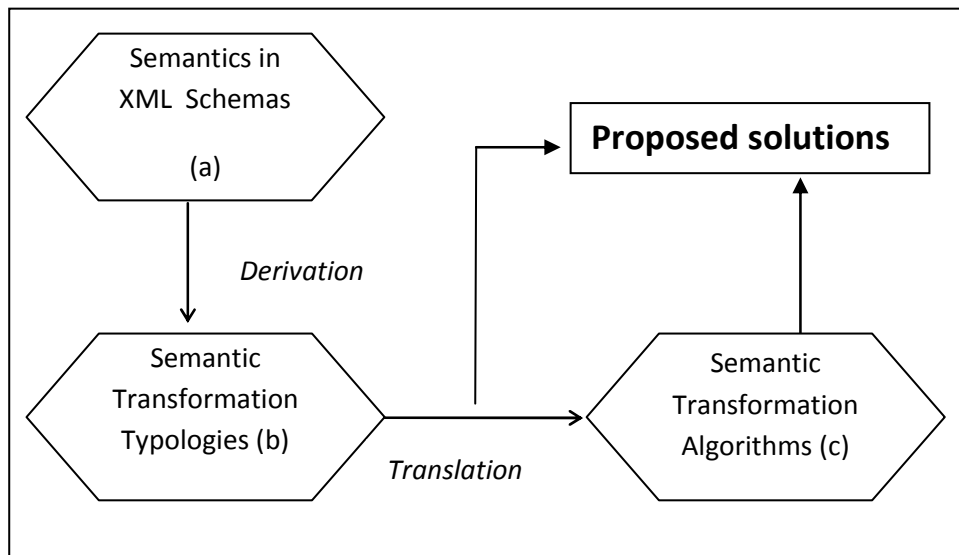


Figure 1.2 Outline of Thesis Scope

The scope of the research is broadly summarized in Figure 1.2. The goals are

- a. to leverage the XML Schema semantics,
- b. to propose a complete typology of semantic transformations, and
- c. to propose algorithms for semantic transformations.

There are two sets of constraints in XML Schema that need to be leveraged for transformation tasks, namely structural constraints and constraints of elements. Structural constraints describe how content is modeled as tree expressions in XML documents. Constraints of elements are those restrictions and values imposed on

each data element. To semantically transform an XML query, the structural constraints are essential constraints, and constraints of elements are optional constraints, which are used depending on the XML query components.

There are three important sets of XPath query components that need to be transformed. These are: (a) XPath query with simple path expression, (b) Path query specified with XPath axes, and (c) XPath query specified with predicates.

- a. For XPath queries specified with simple path expressions such as ***, */*, *//*, several semantic transformations are proposed, whereby each semantic transformation addresses individual path expression.
- b. For XPath queries specified with XPath axes such as *child*, *self*, *parent*, *preceding*, *following*, *preceding-sibling*, *following-sibling*, *ancestor*, *ancestor-or-self*, *descendant*, *descendant-or-self*, several semantic transformations are proposed, whereby each semantic transformation addresses individual XPath axes.
- c. For XPath queries specified with predicates, several semantic transformations are proposed, whereby each semantic transformation addresses predicates that support a single condition, conjunctive conditions and disjunctive conditions in a given XPath query. The condition is a Boolean expression which may involve comparisons between elements and values, path expressions denoting elements to be compared as well as further path expressions. These will be taken into consideration when semantic transformations are proposed.

This thesis is organized into ten chapters. The inter-relationship among the chapters is depicted in Figure 1.3. The contributions of each chapter are briefly described below.

Chapter 1 provides a brief overview of the background of XML. It then briefly discusses XML databases. This chapter also presents the details of XML Schema and its importance in general. XML query processing is introduced in order to highlight the main XML query languages. The motivation for this research is then explained, highlighting the need for this study. This chapter also discusses the objectives and scope of this research.

Chapter 2 reviews previous works that address the semantic query transformation. We provide analyses of various sections of existing works. Each section reviews a particular approach that is adopted by several works. Various weaknesses and strengths of existing techniques will be highlighted to explain the motivation for this research.

Chapter 3 describes the open problems in XML semantic query transformation and the problem areas that are going to be addressed in the thesis. The section addresses the shortcomings of the existing techniques that were highlighted in Chapter 2. The identified problems include the unavailable semantic transformations for certain types of XPath structures, XPath components such as axes, and predicates where conditions are specified for data filtering. This chapter also includes definitions of XML Schema, data model, XPath fragments and related XML essentials required for the semantic transformation methodology.

Chapter 4 consists of two sections:

1. The first section proposes a technique to derive semantics from given XML Schemas to support the proposed semantic transformation typologies. The goal is to minimize the transformation process when semantics are matched and selected by semantic transformations.
2. The second section proposes the first category of semantic transformations. This semantic transformation category proposes three typologies to transform a simple XPath expression that is specified without conditions or axes. An XPath query can be contracted, expanded or complemented based on the XPath fragment used in the XPath query.

Chapter 5 proposes the second category of semantic transformation typologies, that is, for an XPath query specified with axes. The goal of semantic transformations is to eliminate the axes from the XPath query where possible.

Chapter 6 proposes the third category of semantic transformations which are transformations for XPath queries specified with predicates. Due to the complexity of the predicates in XPath query, the transformations are concerned with eliminating

a whole predicate or reducing the size of a predicate. The semantic transformation rules identify the structure in predicates and matching semantics before deciding on semantic transformation.

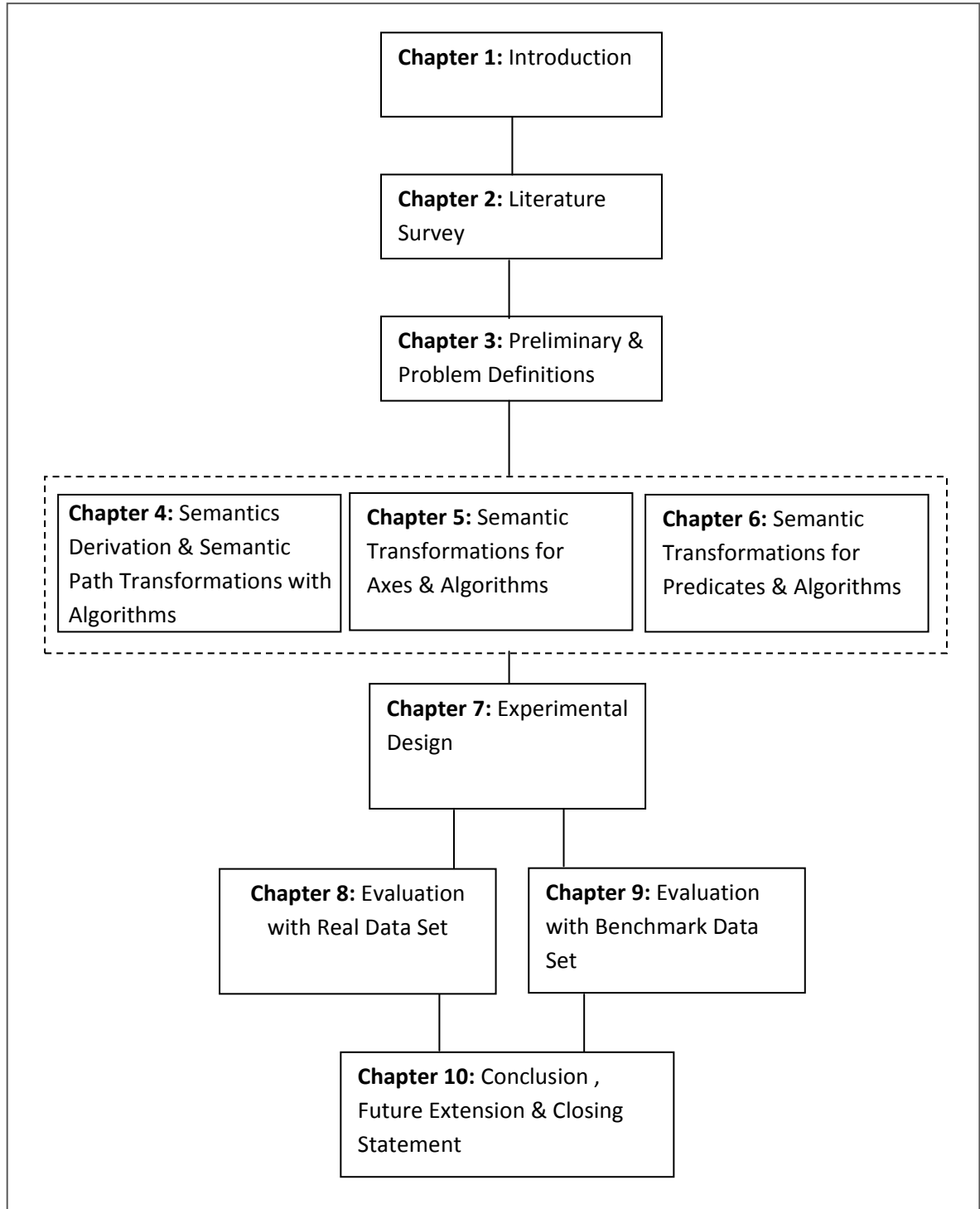


Figure 1.3 Inter-relationships between Chapters in the Thesis

Chapter 7 describes the experimental methodology and design, which include the experimental background, objective and two main strategies. The first describes the common tasks that are shared by the experiments. The second describes the tasks conducted by individual experimentations.

Chapter 8 evaluates and analyses the query performance results of the designed XPath queries using a DBLP real data set [Ley, 2011]. Both DBLP schema and data are studied and analyzed before new semantics are added. To ensure the consistency and quality of the data, a validation tool for data consistency is employed.

Chapter 9 evaluates and analyses the query performance results produced by the benchmark XPath queries and their semantic XPath queries using benchmark data [Runapongsa et al., 2006]. While the real data and its XML Schema may not be comprehensive enough for evaluating the effectiveness of the semantic transformations, benchmarks that provide XML Schema, complex data structures and queries are much more relevant in order to study query components that have an impact on performance. The goal in this chapter is to apply semantic transformations to eliminate the query components that are believed to affect query performance.

Chapter 10 presents an overall summary of the works and future directions of the thesis. The conclusion addresses the contributions in each chapter. Finally, we highlight future work that may arise from this current research.

Chapter 2

Literature Review

The aim of this chapter is to show the achievement of existing works in using semantics from the database schemas for optimization purposes and more importantly, to identify the issues which remain outstanding. Some of the outstanding issues are the central focus of this research.

2.1 Clarification & Classification of Techniques

It is important to understand the terms *semantic query optimization* and *semantic query transformation* in order to establish a semantic transformation methodology for query optimization purposes.

The terms *semantic query optimization* and *semantic query transformation* refer to a common concept that applies constraints in the database schemas for optimization purposes. *Semantic query optimization* is a technique using rules or theories formulated from semantics defined in database schemas for optimization purposes. *Semantic query transformations* fulfill the optimization task through a systematic approach. It first establishes a set of rules to transform queries to semantically equivalent but restructured queries. The semantic transformations are then implemented and empirically evaluated for their efficiency and effectiveness as semantic query optimization devices.

The utilization of semantics in the schemas for query optimization has been introduced in relational, object-oriented and deductive databases. It is important to understand how the semantics in the database schemas bring benefits for query optimization purposes. We emphasize that different databases have different structures and their schemas provide a different set of semantics. Some techniques are specific to certain semantics. We study the techniques in the following categories:

- **Legacy Databases:** In this category, we review several important works of semantic query optimization in Relational, Deductive and Object-Oriented databases. We believe the works from different databases would benefit our research significantly.
- **XML Databases:** In this category, we review techniques that apply semantics in XML schemas (DTD and XML Schema). In this category, the techniques are further divided into several sub-groups.

2.2 Legacy Databases

The principle of semantic query optimization is to use semantic rules to reconstruct a query into an equivalent query to deliver better performance [Haseman et. al., 1999; King, 1981a]. Semantic rules are formulated by applying given database knowledge such as constraints defined in the schema. The question is: how can efficient semantic optimization rules be formulated to reduce the cost of performance?

Before discussing related works in semantic query optimization, a partial database schema given below is used to give a brief overview of several semantic rules used in semantic optimization [Haseman et al., 1999; King, 1981a]. Details of this example can be found in the work of Haseman et al. [1999].

Relations

SHIP(shipname, owner, registry, type, capacity, deadwt)

CARGO(cargo#, ship, cargotype, quantity)

OWNER(ownername, industrytype, assets, headquarters)

Join Paths

SHIP.shipname = CARGO.ship

OWNER.ownername = SHIP.owner

Conditional Reference Constraints

All cargos use available ships

CARGO.ship \subseteq SHIP.shipname

Bounding Constraints

Cargo quantities are bounded by ship capacity

CARGO.quantity \leq SHIP.capacity

Semantic Integrity Constraints (SIC)

- All supertankers have deadweights of 100000 or more and all ships with deadweight exceeding 100000 are supertankers.
SHIP.type = 'supertanker' \Leftrightarrow SHIP.deadwt \geq 100000
- Owners with assets greater than 1 billion are considered to be in the petroleum industry.
OWNER.assets > 1Billion \Rightarrow OWNER.industry = 'petroleum'

Suppose that the queries below are transformed based on the availability of semantic integrity constraints (SIC) to obtain semantic equivalent queries (SEQ). While the first SEQ has fewer clauses than those in first Query, the second SEQ has more clauses than those in the second Query.

Query: SELECT SHIP.shipname, SHIP.owner
 FROM SHIP WHERE SHIP.type = 'supertanker' AND SHIP.deadwt
 > 75000

SIC: SHIP.type = 'supertanker' \Rightarrow SHIP.deadwt > 100000

SEQ: SELECT SHIP.shipname, SHIP.owner
 FROM SHIP WHERE SHIP.type = 'supertanker'

Query: SELECT OWNER.headquarters
 FROM OWNER WHERE OWNER.assets > 1 billion

SIC: OWNER.assets > 1Billion \Rightarrow OWNER.industry = 'petroleum'

SEQ: SELECT OWNER.headquarters
 FROM OWNER
 WHERE OWNER.assets > 1Billion
 AND OWNER.industry = 'petroleum'

Over the past decades, several techniques for semantic query optimization have been proposed. Figure 2.1 summarizes the main differences among the techniques applied to legacy database types. This section reviews several important works to highlight the foundation of the semantic query optimization concept.

King [1981a; 1981b] proposes a query language *sub-class* to optimize *Select-Join-Project* queries for Relational databases. The query language sub-class is built by analyzing the select-join-project query at several levels of detail, along the lines of the plan-generate-test paradigm of artificial intelligence notations. The researcher then integrates semantics, query structure, and query processing knowledge to formulate transformation rules such as restriction elimination, join addition, index and scan structures. The formulated rules are then applied to transform the sub-class query language for optimizing *Select-Join-Project* queries. As a result, the performance of the transformed queries was promising when restriction elimination and join addition rules were applied.

The drawback of this work is that the proposed query language sub-class can handle only a single join between relations in the query conditions. Moreover, this work does not provide a control strategy of transformation iteration. For example, suppose that a new restriction is added to a query condition, and after it passes through to the next iteration, another rule decides that elimination of another restriction is needed. The next iteration decides an additional restriction is needed and so forth; this becomes an endless loop. To limit the cycle of transformations for a given query

based on a set of rules, a controlling mechanism is needed to avoid the endless loop of transformations.

Database	Authors	Approach/concept	Constraint	Drawbacks
Relational	King, 1981a and King, 1981b.	Sub-class language to optimize select-join-project.	Schema Knowledge and data values.	No strategy to control transformations. Could be caught in an endless Loop.
	Shenoy & Ozsoyoglu, 1987.	Theoretic graph-based to identify redundant joins.	Implication integrity and sub-set integrity.	No strategy to control transformation iteration.
Deductive	Chakravarthy et al., 1986a; Chakravarthy et al., 1986b; Chakravarthy et al., 1988 and Chakravarthy et al., 1990	Modifying query based on the obtained residues.	Residues.	-
Object-oriented	Grant et al., 1997	Modifying query based on the obtained residues.	Residues.	Involves extensive steps before a query can be modified.
	Meier et al., 2010	Reasoning redundant components by applying type-based	Type-based semantics.	-

Figure 2.1 Semantic Query Optimization for Legacy Databases

The set of transformation rules in this work will influence our research, although the structure of the XML data model is different from the structure of the relational data model. Semantics in XML schemas are regarded as the superset of semantics in relational schemas. XML schemas support both structural constraints and constraints of elements; these constraints can be useful for semantic transformation techniques such as restriction elimination, reduction and join addition.

Shenoy and Ozsoyoglu [1987] proposed a technique for relational databases to use implied and sub-set constraints to formulate a set of heuristic rules including implied expansion, semantic expansion and semantic reduction. The technique first constructs a graph-based simplification method, which enables an identification of redundant joins and redundant restrictions before the heuristic rules perform the tasks of elimination and reduction. Similar to the limitations in [King, 1981a and King, 1981b], this approach does not have a heuristic strategy to control and select suitable types of transformation.

On the positive side, this approach has provided some directions for the research described here. It has a graph-based representation of join-queries which is useful for deciding which of the redundant conditions in predicates can be eliminated. While XML is a tree data structure, XML Schema is a directed graph that represents the structural constraints. We consider this work to be close to our research, specifically in relation to join-queries where conditions focus on an element comparison value.

Chakravarthy et al. [1990] consolidate all existing semantic query optimization techniques for deductive databases [Chakravarthy et al., 1986a; et al. 1986b; et al. 1988]. By performing consolidation, they iteratively apply deductive rules to relations in order to obtain residues, which are also known as integrity constraints. This consolidation is very effective because, when using residues to modify the queries, the performance of the modified queries is improved tremendously compared with the performance of the original queries.

Grant et al. [1997] employ residues to propose a modification of queries in object-oriented databases for optimization purposes. Their approach involves several steps; for example, each object query is translated to a logical representation and then applied with residues. One of the drawbacks of the technique is that they translated an object-oriented schema to a relational schema to obtain the residues. This means some object-oriented semantics are flattened when the object-oriented schema is converted to a relational schema.

The approaches proposed by both Chakravarthy et al. [1990] and Grant et al. [1997] have motivated us to consider the fact that regardless of the types of databases, modified queries perform better than do the original queries.

Meier et al. [2010] use type-based semantics to optimize queries on the database types such as object-oriented databases and deductive databases. The type-based semantics rely on rigorous first-order logic formalization. The first-order logic allows reasoning of the redundant components and hence, some techniques can be applied to modify the redundant components.

In regards to the evaluation strategy, the authors extend a cost-based model proposed by Deutsch et al. [2006] and develop this further to minimize the union of conjunctive queries with negations (using a constraint base, a type system, and a generic cost function to accomplish this task) [Björklund et al., 2008]. This work utilizes interrelation semantics, which model relationships among the objects. Their captured techniques of interrelation semantics can benefit our research in the modelling of both complex and simple type queries.

In summary, we believe various techniques of semantic query optimization proposed for legacy databases that support optimization rules can be explored for our research. Although the processing and structuring of queries in these databases are different from the processing and structuring of XPath queries in XML databases, certain constraints such as range values of data remain the same for most databases. In addition to this, at the high level of using schema semantics, we can learn techniques such as semantic capturing and concepts and then adopt these techniques in our research. We also learn that by modifying a query, better performance can be achieved in most cases.

2.3 XML Databases

This section reviews existing techniques that utilize semantics from XML schemas for query optimization purposes in XML databases. An XML query (i.e. XPath or XQuery query) can be expressed with or without predicates. We review the existing works in two categories, namely, XML queries without predicates and XML queries with predicates.

Since XML Schema (XSD) has not been fully exploited in semantic query optimization and transformation, before the related works are formally discussed, a

partial of XML schema (DTD) given below is used to give the reader a brief overview of how semantics are used in semantic query optimization.

```
<!ELEMENT publisher (address, book*)>
<!ATTLIST publisher name CDATTA #REQUIRED>
<!ELEMENT address (#PCDATA)
<!ELEMENT book (title, author*)>
<!ELEMENT author (name,age)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT age (#PCDATA)>
```

The schema above implies the *publisher* has an attribute of publisher *name* and multiple copies of *book* as its children. Each *book* has a **title** but many authors denoted as *author* who has a *name* and *age*. These are known as semantics or constraints.

Suppose an XPath query ‘/publisher[@name]//author[age]’ is given. Without the given schema above, the *publisher* and *author* satisfy the attribute *name* and element *age* can be known only when the end tag of *publisher* is reached. Based on the semantics in the given schema above, certain semantic rules can be formulated, which are going to be studied in this section. Given these semantics, both attribute *name* and element *age* are not required when a book *author* is queried. However, the question is whether *book* should be introduced to the XPath query above so that the axis ‘/’ allows the checking of parent *publisher* and child *book* then parent *book* and child *author*. The other option is not to introduce *book* to the XPath query so that the axis ‘//’ allows the checking of ancestor *publisher* then descendant *author*. The answer depends on which option produces better query performance.

2.3.1 XML Queries without Predicates

In this category, the techniques of using semantics in XML schemas (i.e. DTD and XML Schema) for query optimization purposes are further grouped into sub-categories including XPath Query Containment, Tree Pattern Minimization and Semantic Query Optimization. Figure 2.2 summarizes the existing works in this section in terms of approach, constraints used and drawbacks.

Approach	Authors	Constraints	Drawback
XPath Query Containment	Miklau & Suciu, 2002	Semantic equivalences	Not able to find containment if DTD constraints that are equivalent to semantic equivalences used.
	Wood, 2003	Subclass and sibling constraints	Suitable for small scale of data size as limited constraints are applied.
	Wang et al., 2008	Semantics derived from DTD	Not able to find containment for '*' from the equivalent tree pattern
	Zhou et al., 2009 and Wang & Yu, 2010	Parent-child Sibling and cousin	Not supporting wild cards and branching considers non-disjunctive type.
Tree Pattern Minimization	Amer-Yahia et al., 2001, Amer-Yahia et al., 2002	Subtype, required- child and required- descendant	Limited constraints. Show better runtime when subtype constraint is not used
	Ramanan et al., 2002		Only show an improvement when the subtype constraint is excluded.
	Chen & Chan 2008	Required-child, required-descendant, required-parent, required-ancestor, subtype and sibling constraints.	Supports no recursion and requires uniqueness among elements
Semantic Query Optimization	Bohm et al., 1998	Exclusivity, Obligation, and Entrance Locations	Do not directly apply constraints to optimization
	Wang et al., 2003	Schema Paths	Do not target for query performance.
	Su et al., 2005	Path constraints with inference type support	Work targets for stream processing. Do not consider XPath axes
	Che et al., 2006	Contain and Contains-in	Limited number of semantics
	Sun & Liu, 2006	ISA, PartOf and SynOf	Not sharing evaluation results

Figure 2.2 Summary of Existing Works for XML Query Specified without Predicates

XPath query containment determines a set of answers to one XPath query, is a subset of of answers of another XPath query. Several techniques have been proposed to study XPath query containment problems. We review the techniques that apply semantics to find containment for optimization purposes.

a. XPath Query Containment

XPath query containment normally involves XPath fragments such as parent-child, descendant-ancestor, wildcard, and branching. These XPath fragments can be effectively removed and replaced by containments found in an equivalent tree pattern that is represented by an equivalent XPath query.

Miklau and Suciu [2002] propose a set of semantic equivalences to find containment of XPath fragments such as parent-child, descendant-ancestor and branching XPath fragments. Semantic equivalence is a tree-based mapping used to find similarity of semantics detected in the DTD. Semantics derived directly from DTD are referred to as conventional semantics.

By using semantic equivalence between two tree patterns that contain XPath fragments such as parent-child, descendant-ancestor or branching, Miklau and Suciu [2002] try to prove the existence of a homomorphism (a mapping between two tree patterns that respects the whole data tree structure). For example, if the second tree pattern contains some sub-patterns which match more XPath fragments such as parent-child, descendant-ancestor or branching in the first tree pattern in the presence of semantic equivalences, then a homomorphism exists. When semantic equivalence is applied to the two tree patterns, they are verified by applying the semantics (or constraints) in DTD (which are used to derive semantic equivalence); however, a homomorphism cannot be detected. This shows that by using semantic equivalence, deciding the containment of one tree pattern in another tree pattern is not always successful. This adds a limitation to the work since semantic equivalence and constraints (used to derived semantic equivalence) are not fully compliant.

Wood [2003] considers XPath fragments such as descendants-ancestor, wildcard and branching in finding XPath containments. He shows that containments for these XPath fragments can be found by applying a subclass and sibling constraints in DTD. While the result for XPath containment of XPath fragments such as wildcard or branching nodes indicates a polynomial runtime; however, when finding XPath containment for descendant fragments, performance is slow. This solution is suitable for a relatively small scale of data as the number of constraints would be minimal.

Wang et al. [2008] explore the DTD semantics to find query containment for tree patterns that contain XPath fragments such as parent-child, descendant-ancestor, wildcard or branching. The objective of this work is to provide an efficient algorithm that can take in DTDs directly and use the semantics to find containments in a tree pattern represented for an XPath query. Their algorithm is designed to support a wildcard existing in the first tree pattern, and the containment represented for the wildcard is expected to be found in the second tree pattern. The first tree pattern is different in structure from the second tree pattern but they are semantically equivalent. However, it turns out the containment of a wildcard in the first tree pattern cannot be found as a containment in the second tree pattern by the algorithm. Unfortunately, this work does not fully meet its objectives. Therefore, the first tree pattern must be transformed to a new tree pattern before the algorithm can determine the containment of a wildcard in the second tree pattern.

Zhou et al. [2009] as well as Wang and Yu [2010] use a chasing technique of a tree pattern to minimize XPath queries. They identify constraints of the required child, sibling and cousin. These constraints are derived from a recursive DTD, which needs a set of specific chasing rules to derive them. Even though the technique can show the existence of a homomorphism between the tree patterns, a homomorphism is not necessarily needed for the containment of tree patterns with self-or-descendant edges. That is because an improvement in the self-or-descendant edges is not sufficient.

To identify the existence of a homomorphism, the authors consider various special cases. They consider XPath fragments such as parent-child, descendant-ancestor and branching. XPath fragment branching supports only a non-disjunctive type. This work does not consider finding containment for a wildcard, which has been addressed by Wang and Yu [2010] under containment finding.

As techniques proposed for query containments revolve around fragments of child, descendants, wildcard and branching, there are limitations in supporting XPath fragments. Although the work here is not about XPath containments, the semantics using tree-based mapping from XML documents are similar to those semantics

captured from an XML Schema; and we can explore these semantics for our research.

b. Tree Pattern Minimization

A tree pattern is used to represent the structure of XML queries such as XPath query and XQuery. Tree pattern minimization is a technique that expresses an XML query on a tree pattern and minimizes the size of a tree pattern for query optimization purposes.

Amer-Yahia et al. [2001, 2002] apply constraints such as subtype, required-child and required-descendant to minimize the size of a tree pattern that represents a given XML query.

The performance of the execution of a minimized tree pattern is evaluated using two proposed algorithms. The first algorithm augments the nodes and edges, and removes any redundant nodes and edges. The second algorithm propagates the remaining nodes and edges to a new pattern tree to produce the final result. The limitation of the first algorithm is that it is restricted to supporting a small number of constraints to produce the unique equivalent minimal query.

As the runtime shows, the technique used in this work is adequate for XML documents that are not significantly large in scale. However, the runtime fluctuates when the data sizes are relatively large. The technique was later refined by Ramanan [2002], who modifies the algorithms proposed by Amer-Yahia et al. [2001, 2002] to improve the runtime of the proposed minimization techniques with no additional constraints other than subtype, required-child and required-descendant. The mechanism involves combining the constraints and then switching them in computations. However, the result indicates a better runtime when the subtype constraint is excluded from the computations.

The common drawback of minimization techniques proposed by Amer-Yahia et al. [2001, 2002] and Ramanan [2002] is that they involve the same set of constraints; they lack the other common constraints such as required-parent or required-ancestor.

Chen and Chan [2008] propose a technique to minimize XPath queries by considering a larger set of constraints including subtype, required-child, required-descendant, required-ancestor, required-parent and sibling. In their optimization technique, they first categorize the constraints into forward (subtype, required-child, required-descendant and sibling) and backward (required-ancestor and required-parent) groups. Optimization is accomplished by a set of algorithms that provide rules to produce a set of minimum XPath queries or single minimized XPath query for an input XPath query.

Most of the input XPath query is minimized and results in a single minimized XPath query. However, there are cases of an XPath query being minimized and resulting in a set of minimum XPath queries. The reason is that the target element in the input XPath query is a wildcard, which represents a set of different schema elements.

The runtime of a minimized XPath query or a set of minimum XPath queries outperforms the runtime of the original XPath query. Although this work supports more constraints than do any earlier works, it still has a limitation because it recognizes only distinct element types in the structural constraints set.

As most of the techniques proposed for tree pattern minimization use constraints such as subtype, required-child and required-descendant, very few works extend the list of constraints to support required-parent, required-ancestor and sibling. Although the work here is not about tree pattern minimization, the semantics such as subtype, required-child and required-descendants are similar to semantics that can be captured from XML Schema. The semantic terms are referred to as co-occurrence, occurrence or nested type; they are very useful to determine the existence of a child and the children of a child.

c. Semantic Query Optimization

Semantic query optimization is a technique that uses schema semantics to formulate a set of rules and theories, which are then used to optimize XML queries (i.e. XPath queries and XQuery queries).

Bohm et al. [1998] propose an approach using schema semantics to create a structure index for query optimization. This approach first constructs a structured index

(referred to as a structure index) based on a set of constraints including exclusivity, obligation, and the entry locations. It then uses the structure index to speed up query processing. Using the structure index to optimize the query is regarded as a logical optimization, which is usually built directly on the database.

Due to the use of an index, the drawback of this approach is that the storage details need to be known in order to make plans for query execution. The necessity of having knowledge about the storage details defeats the purpose of *semantic query optimization*. *Semantic query optimization* does not need to know about the storage details and system resources. It only requires information such as schema semantics.

Wang et al. [2003] are concerned with two techniques, *path shortening* and *path complementing*, to rewrite XPath queries using schema paths

- A *path shortening* technique rewrites a single path to a set of minimum path expressions. They consider only simple path expressions such as parent-child '/', descendant-ancestor '//' or wildcard '*'. The single path does not indicate the use of an XPath axis such as following, preceding, following-sibling or preceding-sibling.

Their main intention is to evaluate memory efficiency instead of query performance for optimization purposes. Each minimum path expression in the set of minimum path expressions is executed to produce one schema node, which produces a set of data nodes at a time. The aggregate result produced by the set of minimum path expressions is able to improve memory efficiency.

- The *path complementing* technique tests the aggregation of outputs to complement the path shortening technique. It does not test for memory efficiency or query performance. Currently, it does not clear the buffer before the next minimum path takes over. For example, a query asks to return all names in a given region, '//region//name'. In the schema, suppose that there are two path expressions that compute information 'name'. The first expression produces the item name in all regions, '//region/item/name' and the second expression produces the name of the people in all regions, '//region/people/name'. It executes the first expression and then the second

expression. The final result is the first result aggregated with the second result.

The objective of this work is different from our objective here. We complement this work and acknowledge the technique of path shortening to produce memory efficiency.

Su et al. [2005] propose a semantic query optimization for XQuery for stream databases. They use a query tree to capture a path constraint and then apply an inference type that indicates a sequential order among the paths to resolve recursion and descendant issues. In addition to path constraints, they also use constraints such as order, occurrence and inclusivity. By using these constraints, they can identify the redundant components in XQuery queries that do not contribute to the final result.

This work focuses on semantic query optimization specific to stream databases. In XML stream processing, information retrieved is in a token that generally causes a memory buffering issue. One of the critical issues for XML stream processing in this work is a change of schema. They have not provided a mechanism for handling the changes in a schema. In addition, they do not address optimization of XPath fragments such as parent '..', as they are concerned mainly about wildcard '*' and descendant-ancestor '//'. This work applies semantics derived directly from the schema to rewrite XQuery queries. In a similar direction, here we derive semantics directly from XML Schema and apply them in our transformations of XPath queries.

Che et al. [2006] propose a heuristic-based algebraic technique for a query optimization framework. The heuristic-based algebraic notations are a set of PAT algebraic expressions [Salminen & Tompa, 1994], which are developed for structured text access. PAT algebraic expressions allow the checking of type consistency and consequently produce constraints, namely 'contain' and 'contain-in'. The produced constraints are used to support their heuristic-based algebraic transformation. Although the main technique proposed by this work does not focus on semantic query transformation, the authors consider the use of constraints in schemas such as DTD or XML Schema when available. Hence, they focus on the use of schemas as additional resources for further optimization at the application level. In this work, semantics are not the standard ones that can be directly derived from

DTDs. They are specific semantics such as contain and contain-in that need PAT algebraic expressions for derivation. Their heuristic-based algebraic technique is influenced by the specific constraints.

Sun and Liu [2006] apply an ontology conceptualization to improve query performance for optimization purposes. They utilize the hierarchical organization of concepts such as inheritance ("ISA") relationship, part-whole ("PartOf"), concept-value ("ValueOf") and synonym ("SynOf") among many others to formulate a set of rules. These rules transform normal XML queries to semantically equivalent XML queries, which have a query runtime that is less than the query runtime of the original XML queries.

Ontologies are useful for capturing the semantics of a data source and to unify the semantic relationships between structures. From this work, we see an opportunity for our future research so that it can support semantics derived from data sources. As for this research, we do not consider semantics in data sources. We consider semantics derived from an XML Schema.

We believe this work can be enhanced to provide a practical solution as currently no performance evaluation study has been undertaken. One way to enhance this is to provide additional rewriting rules to translate the rules derived from ontologies. The translated rules can be implemented for experimentation. In this way, the technique can fundamentally provide the value of semantic query optimization from the ontological aspect.

In this section, we have studied techniques of *semantic query optimization* for query optimization purposes. We find that some but not all, of the existing works use semantics derived directly from a DTD. Some existing works do not use DTD semantics directly; instead, they use specific constraints that need other techniques to derive them [Bohm et al., 1998; Che et al., 2006]. The main outstanding issue is that none of the existing works has addressed semantics in XML Schema for optimizing XML queries. We believe XML Schema will improve the techniques of using semantics for optimization purposes as there are many features that are not supported in DTD.

2.3.2 XML Query with Predicates

Figure 2.3 provides a summary of existing works in terms of constraints used and their limitations.

Authors	Constraints	Drawback
Fernandez & Suciu, 1998	Schema graph	Supports only single conditions
Kwong & Gertz, 2002 and Olteanu et al., 2002	Parent-child, sibling, order and semantics from DTD	Supports only nested conditions Supports no joins of conditions
Su et al., 2005	Occurrence, exclusive, inclusive and enumeration	Cannot modify conditions with value-based types due to the use of DTD
Wang et al., 2006	Occurrence, inclusive and exclusive	Provides only management plans on execution of predicates
Groppe & Böttcher 2005	Schema graphs	Considers detection of semantic conflicts in queries as optimization solutions.
Bao et al., 2008	Semantics from DTD	Cannot support conjunctive predicates. Need to break up the conditions
Li et al., 2008	Occurrence, inclusive, exclusive and pattern non-occurrence	Cannot support disjunctive predicate. Conjunctive predicate cannot be modified or removed if comparison is on a value-based type.
Hanson & Mani, 2010	Parent-child and ancestor-descendant	Translate XQuery to SQL/XML syntax which modifies expressions of parent-child or descendant pairings. The work is a preliminary study of simple XQuery that is limited to traditional database management systems.
Wu et al., 2010 and Wu et al., 2011	Structural semantics and value-base semantics of elements	Extract semantics from XML documents and store into relational tables. Use them to optimize twig pattern queries. As semantics are stored in relational tables, the impact is that they may be flattened once they are converted to a relational structure.

Figure 2.3 Summary of Existing Works for XML Query Specified without Predicates

A predicate in an XML query expresses conditions to be fulfilled in addition to a structural path. A condition is a Boolean expression. It may involve comparisons between elements and values, path expressions denoting elements to be compared, as well as further path expressions. We refer to predicates in XML as XML query

predicates. The existing work on the use of XML schemas (DTD and XML Schema) semantics for optimizing XML queries specified with predicates is insignificant.

Fernandez and Suciu [1998] propose a query rewriting technique which they refer to as *state extents*, based on a schema graph. The schema graph has a set of elements regarded as states noted as $s_1, s_2, s_3, \dots, s_n$ called *states* in which s_1 is the root and edges are labelled with unary predicates.

The rewritten query using *states*, handles a single condition by breaking it into sub-conditions. Each sub-condition is then processed as a sub-tree. Each sub-tree has a target node. When searching for a target node, the sub-tree searches in a portion of an XML document instead of the whole XML document. This not only reduces the search space, but also improves the overall query performance. The weakness of the technique is that due to the state extents, a query can have only single condition; hence, no joins of conditions are supported.

Kwong and Gertz [2002] and Olteanu et al. [2002] provide a technique to optimize XML queries by using semantic equivalences which are derived from DTDs. The technique first formulates a set of XML query-algebra proposed by Bohm et al. [1998] to establish a framework to modify complex nested XML query predicates. The framework starts off by identifying the redundant conditions in an XML query predicate using semantic equivalence. Some semantic equivalences are location paths. Subsequently, the location paths are the ones that determine whether or not the nested path expression is redundant.

This work focuses on nested conditions in XML query predicates, instead of using sub-paths to express conditions. Although it is very rare for XML queries to adopt a nested path expression for conditions, it is good to be aware of the different condition structures which create further opportunities to study query performance for predicates. The limitation of this work is that it does not support the joined conditions in predicates.

Su et al. [2005] and Li et al. [2008] propose a semantic query optimization technique for XML queries specified with predicates in XML stream databases using semantics in DTD. While Su et al. [2005] address predicates that support single conditions

using occurrence, inclusive and exclusive constraints, Li et al. [2008] address predicates that support conjunctives using a non-occurrence pattern constraint; these two works achieve different objectives. The former tries to cope with the buffering issue that leads to a query performance issue; the latter tries to address the footprint memory issue, which leads to a memory efficiency issue. Regardless of their objectives, they use schema semantics to achieve their goals.

The techniques have several drawbacks. For a single condition, Su et al. [2005] can support only a condition without a value-base comparison. For example, if a single condition is [zipcode > 3004] their technique is inadequate for checking the range value due to the lack of data types in DTD. The same applies to conjunctive conditions [Li et al., 2008] when conditions are joined. For example [zipcode > 3004][news/date], if the value 'zipcode' does not exist in the database, the technique cannot determine this conflict. Hence the query needs to be sent to the database for confirmation. This problem is caused by the limitation of data types supported by DTD. Semantic query optimization for this kind of requirement can be achieved using XML Schema because constraints on data types are available. This shortcoming will be addressed in this research.

Groppe and Bottcher [2005] provide a schema-based approach to study XPath satisfiability and optimization of XML queries. Their approach elaborates on an ordered schema graph generated from XML documents. The ordered schema graph is produced based on three constraints including parent-child, sibling and sequence or choice between the elements. The input XPath query is transformed into an ordered query graph which is evaluated against the ordered schema graph for deciding if the query conditions affect the whole result set, which is either 'no' or an empty result; this becomes the solution to optimize XML queries. Their work can be considered as a work in the semantic query transformation, as their solution is dependent on the semantic conflicts detected in a query. No further modification of valid XML queries is made to reduce the processing of query components as consequently it improves query performance for optimization purposes. This work will influence part of our contribution as their detection of semantics is useful for our work.

Wang et al. [2006] propose a technique to trigger schema runtime information for improving the static semantic query optimization technique proposed by Su et al. [2005] in stream databases. In this work, Wang et al. [2006] provide a runtime management plan to rewrite the algorithm provided by Su et al. [2005] to avoid data buffering by an early detection of predicates and switching the output mode to compute the result immediately.

Their work focuses on memory efficiency as processing consumption for stream databases can be intensified due to the pattern retrieval concept and the volume of processed data being significantly high. If the switching of the output mode works well, then numerous resources can be saved for other useful tasks. In addition, their work does not provide techniques for optimizing the XML query; instead, it provides an online management plan to manage execution based on the detected predicates. Therefore, we may conclude that the conditions they support in predicates still remain as single element conditions, which is supported by Su et al. [2005] However, we acknowledge this technique and will incorporate it in our future work to deal with memory efficiency.

Bao et al. [2008] explore Object-Relationship-Attribute models (ORA-SS) [Wu et al., 2001] to store semantics that are captured from DTDs for query optimization purposes. The authors address three types of predicates. The first type is a single condition with a value, the second type is a single condition with no value and the third type is a disjunctive condition (referred to as a twig pattern) in predicates. Bao et al. perform a query breakup to distinguish the processing of parts of twig patterns in a predicate. They find the first twig pattern match and ignore the rest as they regard the rest of the twig patterns as redundancies.

Due to this query breakup and processing of parts of twig patterns, the approach is limited when addressing conjunctive predicates. When a conjunctive predicate exists, each pattern in the predicate must be processed because conjunctive predicates allow some conditions to be true and some not to be true. As a consequence, the result of true conditions must be produced. Bao et al. [2008] presented the result which indicates a promising query performance improvement. This technique is evaluated by comparing its query performance results against the

query performance result produced by TwigStack [Bruno et al., 2002]. In addition, the technique does not evaluate various data sizes. With various data sizes, the query performance can demonstrate scalability and effectiveness of the approach.

Hanson and Mani [2010] explore opportunities to utilize two structural constraints such as parent-child and ancestor-descendant semantics to modify XPath expressions in XQuery. By modifying an XPath expression, they investigate the pairings of parent-child or ancestor-descendant and remove the pairs that they believe to be redundancies. The new XQuery is then translated to SQL/XML syntax for traditional database management systems.

Semantics in XML Schemas are useful in many areas including query processing. Currently, there are no adequate strategies for optimizing XPath queries by utilizing semantics defined in XML Schemas.

Due to the complexity of XML query predicates, we have seen several techniques which have been integrated with other techniques. For example, Bohm et al. [1998; Kwong and Gertz [2002] and Olteanu et al. [2002] derive semantic equivalence instead of using semantics directly from XML schemas.

Wu et al. [2011] construct relational tables that store semantics extracted from XML documents. They use the semantics in a table to optimize a twig pattern query by avoiding process patterns that make no contribution to the final result. This work which extracts semantics from XML documents and stores them in relational tables would definitely face challenges, the obvious one being that the referenced path by ID cannot be identified easily. Due to this problem, the work considers an extension to using ID references in DTDs to improve the processing of referenced paths [Wu et al., 2010]. We currently use semantics in XML Schemas to provide a transformation strategy to find opportunities to optimize XPath queries. However, the work of Wu et al. [2011] can be integrated into our future work on capturing semantics from XML documents. The captured semantics can then be combined with semantics provided in XML Schemas and DTD for a complete solution to semantic query transformation.

In summary, we have reviewed several works that apply different techniques using semantics from the schemas to optimize XML queries specified with predicates. The existing works support different types of predicates that in turn support single conditions with or without values, join conditions and nested conditions. As indicated by the literature review, most of the existing work focuses on conjunctive predicates using semantics for optimization.

2.4 Summary and Open Challenges

In this literature survey, we have reviewed several major techniques for semantic query optimization for XQuery and XPath queries using constraints from XML schema. We have provided insights into individual techniques and studied the constraints used by individual techniques to optimize XML queries.

Initially, we explained the difference between the terms *semantic query optimization* and *semantic query transformation*. After that, a review was conducted of the semantic query optimization techniques in legacy databases including relational, object-oriented and deductive databases. We also reviewed the optimization techniques relevant to our research.

After examining legacy databases, we reviewed the techniques that used semantics to optimize XML queries. We divided the techniques into two categories of queries: XML queries without predicates and XML queries with predicates. In the first category, XML queries without predicates, we group the techniques into three groups including *XPath Query Containment*, *Tree Pattern Minimization* and *Semantic Query Optimization*. In the second category, XML queries with predicates, we reviewed the existing works that focus on techniques and the types of predicates that these techniques support. We found that few works addressed disjunctive conditions in predicates, which has become one of the major outstanding issues for the research presented here. Based on the summary above, we observed the following:

- There are no works that propose a *semantic query transformation* technique which can be evaluated systematically so that each semantic query transformation can be identified as an optimization device. Most of the

existing works focus on semantic query optimization which is different from *semantic query transformation*.

Semantic query optimization is a technique that targets query optimization directly by using rules and theories to optimize XM queries. *Semantic query optimization* techniques do not go through systematic evaluation.

Semantic query transformation on the other hand first transforms an XML query to a semantically equivalent query by using semantics derived from XML schema. The semantic query transformations lead to optimization after being systematically evaluated and able to produce optimized query performance results.

- Most of the existing works address semantic query optimization techniques using semantics derived from DTD. Works using semantics derived from XML schema for optimization purposes require much attention; this is important because there are many features that are available in XML Schema but not in DTD [W3C, 2004a; 2004b].

In particular, the new features are useful for transforming XML queries specified with predicates. For example, conditions in predicates have comparison values and are based on various data types. The values cannot be verified as atomic data types such as integer or date and many more are not supported in DTD. Now that the data types are enhanced in XML Schema, there is an increase in opportunities to use semantics for handling predicates for query optimization purposes.

- Apart from simple XPath expressions and XPath predicates, XPath axes including *child*, *ancestor*, *parent*, *self*, *descendant*, *following*, *preceding*, *namespace*, *attribute*, *ancestor-or-self*, *descendant-or-self*, *following-sibling*, *preceding-sibling* provide different navigations of XML documents. Most of these axes, except namespace which provides a means to differentiate elements, allow XML information to be navigated in various directions. None of the existing works have addressed the semantic query optimization for the complete set of axes even by using semantics in DTD.

Today, more database vendors provide excellent techniques such as object-relational, binary, native and many more, for managing and storing XML information. However, query processing still focuses on simple path expressions such as child '/', and descendent '/' [Liu and Murthy, 2009; Zhang et al., 2009]. This means the processing of XPath axes such as following, preceding, following-sibling, ancestor, or preceding-sibling among others, has not yet been improved for new XML storage management techniques.

Chapter 3

Problem Definitions

Chapter 2 presented a literature review that led to the proposed research on semantic query transformation. The review discussed existing works that focus on semantic query optimization in several types of databases including legacy, object-oriented, deductive and XML databases. Light was also shed on a number of outstanding issues related to using semantics for optimizing queries. In this chapter, a problem definition is given to form the basis of a methodology which will be utilized to develop techniques to solve the currently identified problems.

This chapter defines and describes the problems that we are going to address and resolve in this thesis. In order to do this, it is necessary to first provide overviews of XML technology which relates to documents and the XML Schema, XML query structure and query processing concept. The problem is then defined. We also discuss our proposed choice of techniques.

3.1 An Overview of Problem Definition

With the increase in popularity of XML technology, XML Schema has become a better choice due to its richness of semantics and greater flexibility with regard to data structures [W3C, 2004a; 2004b].

Due to the latest development of XML Schemas, the database developers can exploit the great advantage of semantics defined in the XML Schema for transforming XPath queries for facilitating query optimization.

Figure 3.1 provides an overview of two groups of XML Schema semantics including structural constraints and constraints of elements and their usefulness for our proposed semantic transformations. The main tasks of the semantic transformations are:

- to make sure XML Schema semantics are fully utilized for establishing a transformation methodology, to transform XPath queries to equivalent XPath queries for optimization purposes.
- to be able to identify unsatisfied XPath queries so that they are not sent to databases that may incur a high cost of resource usage.

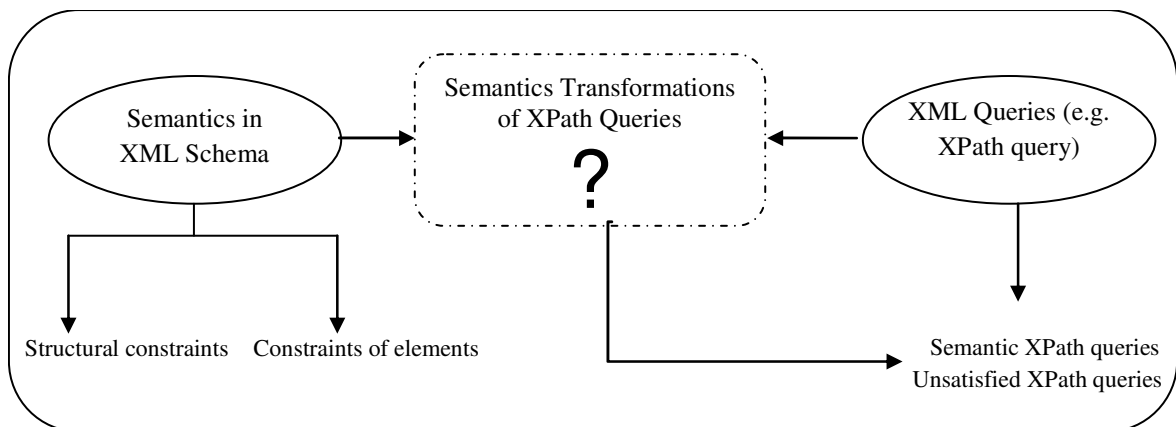


Figure 3.1 Overview of Semantic Transformation Methodology

Based on the semantic transformation methodology as illustrated in Figure 3.1, the semantics (constraints) defined in XML Schema will address the following semantic transformation categories and issues:

- The transformations of simple XPath expressions, which are specified with no query conditions, to equivalent XPath queries.

XML information can be navigated with different path expressions with simple hierarchy relationships such as ancestor-descendant or parent-child relationships

between the elements and wildcard expressions. Unfortunately, some simple path expressions may lead to performance issues. Wildcard operators ‘*’ and ‘//’ are commonly used in XML query [W3C, 1999; 2007a; 2007b; 2010] and are well-known for contributing to performance issue during query processing [Bashir & Boulos, 2005; Deutch et al., 2006 and Wood, 2003]; they should be avoided if possible. It is advisable to transform all possible different path expressions so that those that affect query performance can be avoided.

- The transformations of XPath queries which are specified with XPath axes such as *ancestor*, *parent*, *self*, *descendant*, *following*, *preceding*, *namespace*, *attribute*, *ancestor-or-self*, *descendant-or-self*, *following-sibling* or *preceding-sibling*.
- Since XPath axes [W3C, 1999; 2007a; 2010] are W3C standard axes, they play very important roles in allowing navigation information for different purposes. Some of them do have optional path operators, which make it easier to access non-native XML databases. However, there are a number of XPath axes that do not have optional path operators; therefore, there are significant challenges in both processing and performance. This is due to the fact that some XML-Enabled databases are still unable to provide full support for some of these axes. It is necessary to transform XPath queries specified with these axes to equivalent semantic XPath queries which enable smooth query processing and better query performance.
- The transformations of XPath of XPath queries which are specified with predicates. A predicate in an XPath query expresses a condition to be fulfilled in addition to the structural constraint imposed by the path itself. The condition is a Boolean expression. It may involve comparisons between elements and values, path expressions denoting elements to be compared, as well as further path expressions. Since predicates can accommodate different types of conditions, this may lead to complexity issues. It is important to address the differences in transforming each type of condition; therefore, semantic transformations for XPath queries specified with predicates are needed.

The first two categories of semantic transformations associate their semantics with structural constraints such as the *hierarchy relationships* of elements, the occurrence

of elements within another element or in XML Schema as they deal with path expressions and XPath axes.

The last category of semantic transformation associates their semantics with both structural constraints and constraints of elements. As described, a condition may involve a comparison between elements and values, or path expressions. In the presence of a value comparison, the semantic transformation requires constraints of elements such as *enumeration*, *pattern*, *inclusive*, *exclusive* and many others. We will discuss the features and roles of these semantics later in this chapter.

3.2 XML and Query Essential Background

This section first introduces the basis of structures related to XML documents, schemas and the data model such as query components and terminologies; it then addresses the notions of XML query processing.

The XM Schema definition (XSD) describes the structure of XML documents or databases. For reasons of consistency, the term XML node is going to be used throughout this thesis. An XML node can be either an attribute or element. An XML element contains everything within the beginning and ending tags. The definitions in relation to XML schema and documents for this research concerns the structures of both XML schemas and documents and not other XML related areas such as tree grammars[Murata, et al., 2005].

Definition 3.1. (XML Schema Structure). An XML Schema structure \mathcal{S} is a rooted tree graph \mathcal{G} that is represented by $\{L, E, r\}$ where

- L is a non-empty set of labelled nodes
- $E \subset L \times L \times N \times (N \cup \{\infty\})$ where
- $N = \{1, 2, 3, \dots\}$ and E is a set of edges associated with a multiplicity (j, k) Such that $\langle l_1, l_2, j, k \rangle \in E$. j and k represent the ordering of the nodes l_1 and l_2 for the edge determined by a depth first search of the corresponding XML Schema S where $j < k$
- $r \in L$ where r is the root node of S

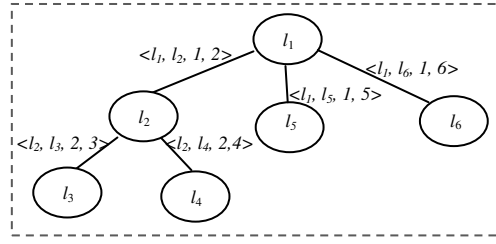


Figure 3.2 Example of XML Schema Tree

Figure 3.2 shows an example of XML Schema structure where $L = \langle l_1, l_2, l_3, l_4, l_5, l_6 \rangle$, $E = \{ \langle l_1, l_2, 1, 2 \rangle, \langle l_2, l_3, 2, 3 \rangle, \dots, \langle l_1, l_6, 1, 6 \rangle \}$, $r = l_1$, $N = \{ 1, 2, 3, \dots \}$, l_1 and l_2 are associated with $(j, k) = (1, 2)$, l_2 and l_3 are associated with $(j, k) = (2, 3)$ and so on.

An XML document provides hierarchical structure to organize nodes with respect to their contents. Any XML document associated with given schemas means the XML document conforms to structures described in the schemas.

Definition 3.2. (XML Document Structure). An XML document structure is a rooted tree denoted by $T = \{N, \mathcal{E}, r\}$ where

- N_o is a set of nodes
- \mathcal{E} is a set of directed edges
- $r \in N_o$ is the root node

An XML document structure T conforms to an XML Schema structure S if and only if the labelled structure of T corresponds to XML schema structure S .

An XML query such as an XPath query needs to consider the scope of query components and types of paths.

3.2.1 XML Query Component and Structure

As our research focuses on XPath query, it is important for us to discuss the important components of XPath queries such as XPath axes. XPath queries are essentially composed of a succession of axes defining the navigation from a current context node. We summarize how each axis navigates information; details of XPath axes can be found on W3C [1999; 2007a; 2010]. Below, we discuss eleven XPath

axes that are going to be used in XPath queries and transformed for optimization purposes.

- The *child* axis navigates information of the children of the context node.
- The *descendant* axis navigates information of all the descendants, such as the child, the child of a child and so on, of the context node. The descendant axis does not navigate an attribute or a namespace.
- The *descendant-or-self* axis navigates information of all the descendants, such as the child, the child of a child and so on, of the context node. The descendant-or-self axis navigates attributes if there are any.
- The *self* axis navigates information of the context node itself.
- The *parent* axis navigates information of the parent of the context node.
- The *ancestor* axis navigates information of the ancestors of the context node. The ancestors of the context node are the parent of the context node and the parent's parent and so on. The ancestor axis includes the root node, if the context node is not the root node.
- The *ancestor-or-self* axis navigates information of the context node and ancestors of the context node. The ancestors of the context node are the parent of the context node and the parents of parents and so on. The ancestor-or-self axis includes the root node.
- The *following* axis navigates information that occurs right after the context node begins and information traverses along the edges all the way down to the lowest level of the schema.
- The *preceding* axis navigates the information that occurs before the context nodes end and information traverses all the way along the edges back to the root.

Both the *following* and *preceding* axes selection excludes the attributes and descendants.

- The *following-sibling* axis navigates information of following siblings occurring on the right of the context node; it traverses horizontally to the far-right sibling element of the context node. If the context node is an attribute, the preceding sibling is empty.
- The *preceding-sibling* axis navigates information of preceding siblings that occur on the left of the context node; it traverses horizontally to the far-left sibling element of the context node. If the context node is an attribute, the preceding sibling is empty.

Among the XPath axes family, *child*, *descendant*, *parent* and *self* can be optionally specified using the path notations $\{/,//,...\}$ which have been commonly used. XPath axes such as *descendant-or-self*, *ancestor*, *ancestor-or-self*, *following*, *following-sibling* *proceeding*, and *preceding-sibling* have unique functionalities which provide different required information, and these XPath axes do not have optional operators.

The performance of queries denoting the same result by means of different axes may significantly differ. The difference in performance can be affected by some axes, but this can be avoided and will be addressed in this research. The next important component of an XPath query is the predicate [Diao et al., 2003]. Below, we define the XPath query predicates and also show the complexities in XPath query predicates.

Definition 3.3. (XPath Query Predicate). An XPath query predicate is a component that contains query conditions specified for filtering information. It is enclosed with $[]$.

The XPath query predicate filters a node-set with respect to an XPath axis to produce a new node set. The query condition in the predicate is evaluated with elements/nodes in the node-set as the context nodes/element. If the query predicate evaluates to true in any given node, then it is kept in the resulting node set [W3C, 1999].

The scope of predicates in an XPath query is now illustrated. A predicate is composed of one or more of the following:

- binary operators ('=', '!=', '<=', '<', '>' '=>')
- connectives ('OR', 'AND')
- a constant
- path with/without constants
- Context position index function such as position()

Samples of predicates in XPath query forms, where e denotes elements and z denotes constant values:

- $e_0/e_1/e_2/e_3[z_0]$ - A predicate, which contains only a constant value.
- $e_0/n_1[e_{11}]/e_2[e_3/e_4]$, $e_0[e_{11}]/e_2[e_3[e_4]]$, $e_0/*_1/e_2[e_3/e_4]$ - Predicates, which contain path fragments e.g. $[e_3/e_4]$, nest path e.g. $[e_3[e_4]]$.
- $/e_0/e_1/e_2[e_3 > z_5]$ - A predicate, which contains both a path fragment and a constant value.
- $/e_0/e_1/e_2[e_3 > z_5 \text{ or } e_3/e_4 \text{ and } e_3 > z_3]$ - A predicate, which contains path fragments and constant values and disjunctive as well as conjunctive operators.
- $/e_0/e_1 [position()>z_0]$ - A predicate, which contain both a context position index function and constant value.

Definition 3.4. (Full Path). A full path p_{full} is a path expression that consists of a set of nodes that can be an attribute or element and hierarchy '/' or '/' and contains no predicate.

$$p_{full} ::= a/b \mid a//d \mid * \mid n \mid .$$

a , b and d are nodes requiring to be tested, hence are referred to as a node test [W3C, 1999; 2007a; 2010]. '/' denotes a child axis, '/' denotes a descendant axis, '*' represents an arbitrary element, n denotes specific element that has a label $l \in L$,

where L is a non-empty finite set of labels, and '.' is the current node. The set of full path expressions is p_{lin}

Definition 3.5. (Partial Path). A partial path p_{part} is an extension from p_{full} that p_{part} is defined as follows:

$$p_{part} ::= a/b \mid a//d \mid a[q] \mid * \mid n \mid .$$

a , b , and d are elements

$$q ::= a \mid a//d \mid a \text{ AND } b \mid a \text{ OR } b \mid a, b \text{ } o_p \text{ } v \mid n \text{ } o_p \text{ } v \mid \text{NOT } (n) \mid f_n \text{ } o_p \text{ } v \mid f_n \text{ } o_p \text{ } f_n$$

$$o_p ::= > \mid < \mid \leq \mid \geq \mid !=$$

$$f_n ::= position() \mid last()$$

$$v ::= string \mid float \mid int$$

$$n ::= \text{constant integer}$$

$$[] ::= \text{predicate contains a set of conditions } q$$

Definition 3.6. (Location Step). A location step l_s is a navigational step that is composed of three components including predicate (optional), test element and axis (optional).

$$l_s ::= \alpha :: n \mid \alpha :: n [q] \mid \alpha :: * \mid \alpha :: * [q]$$

$[q]$ is optional where $q ::= a \mid b//d \mid a \text{ AND } b \mid a \text{ OR } d \mid a \text{ } o_p \text{ } v \mid n \text{ } o_p \text{ } v \mid \text{NOT } (n) \mid f_n \text{ } o_p \text{ } v \mid f_n \text{ } o_p \text{ } f_n$ where

$$o_p ::= > \mid < \mid \leq \mid \geq \mid !=, f_n ::= position() \mid last(), v ::= string \mid float \mid int$$

$\alpha ::= \text{child} \mid \text{descendant} \mid \text{parent} \mid \text{following} \mid \text{following-sibling} \mid \text{preceding} \mid \text{preceding-sibling} \mid \text{self} \mid \text{ancestor} \mid \text{descendant-or-self} \mid \text{ancestor-or-self}$

n denotes a specific element that has label $l \in L$, where L is a none-empty finite set of labels.

Both full path and partial path are formed by a set of location steps where the axis is solely inclusive of a child, which can be optionally specified; that is, when a location step is specified with no axis. As shown in **Definition 3.6**, a location step other than a child axis supports a series of axes to navigate information for different purposes.

Figure 3.3 provides a set of location steps derived from p , axes, node-test and predicate that make up a full location step, which consequently makes up a location path by the sequence of location steps.

Let us consider an example to demonstrate the location path, location step and other XPath query components.

$$p = \text{descendant::a/child::b[position()= 1]/child::*}$$

Location Step	Axis	Node Test	Predicate
1,2	Descendant, child	a,b	
3	Child	*	
2			[position()= 1]

Figure 3.3 Location Paths

Definition 3.7. (Target Element): A target element is an element that is located as the right-most element in XPath query p .

Definition 3.8. (Non-Target Element): A non-target element is any element except the target element, in XPath query p .

Definition 3.9. (Target Location Step): A target location step is the right-most location step in XPath query p . If p contains only one location step, then the location step is the target location step.

Definition 3.10. (Non-target Location Step): A non-target location step is any location step, except the right-most location step, that is located anywhere in XPath query p .

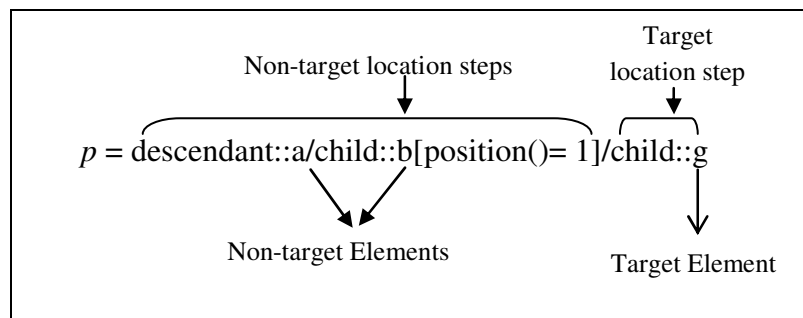


Figure 3.4 Non-target Elements and Location Step, Target Element and Location Step

The example in Figure 3.4 shows non-target elements and non-target location step, target element and target location step in XPath query p .

3.2.2 Notions of XML Query Structural Processing & Equivalence

An XPath expression is a tree pattern in a database [Al-Khalifa et al., 2002; 2002; Yao & Zhang, 2004]. We provide the essential background for XML query processing below.

Definition 3.11. (Tree Pattern). A tree pattern is a labelled tree $\sigma = (V, E)$, where

$V = \{v_1, v_2, v_3, \dots, v_n\}$ is the vertex set, $1 \leq n$

$E = \{e_1, e_2, e_3, \dots, e_m\}$ is the edge set. Each edge e_i is represented by a pair of v , $1 \leq m$

An edge can be a child edge representing the *parent-child* hierarchy denoted as ‘/’ or a descendant edge representing the *ancestor-descendant* hierarchy denoted as ‘//’.

A match of tree pattern σ , a smaller tree than T in a rooted node labelled document tree $T = (V_T, E_T)$, is a total mapping of

$$f: \{u: u \in \sigma\} \rightarrow \{x: x \in T\}$$

For each node $u \in \sigma$ is satisfied by $f(u)$ and each edge (u, v) in σ , $f(v)$ is the child or descendant of $f(u)$ in T .

We hereby define the notion of equivalence with regard to path expressions over the XML documents (database) and XML Schema [Paparizos, et al., 2004; 2007].

Definition 3.12. (Path Equivalence). Two paths P and Q expressed over an XML document T are equivalent if and only if one is a subset of the other and vice versa. That is $P \equiv Q$ iff $[P] \subseteq [Q]$ and $[Q] \subseteq [P]$ where $[P]$ denotes the result set of P and $[Q]$ denotes the result set of Q .

Based on Definitions 3.11 and 3.12, we reason that two paths are equivalent if and only if they both yield the same result set but are different in patterns; that is because

the occurrence of a node in one path may be the subset of the same node in the other path and vice versa.

Without the schema knowledge, such confirmation is obtained from a full processing of paths on a given database to validate the results. Such a procedure can be very costly. With the schema knowledge we can confirm two paths are only semantically equivalent but produce the same result set.

Definition 3.13. (Schema Path Equivalent). Two paths P and Q are schema paths equivalent if and only if one is a super set of the other. Assume Q is super set of P that $P \equiv Q$, then

- All nodes in P map to some or all nodes in Q
- P and Q must end at the same node
- Start node in P can be any node but must occur within and match to the start node in Q

Based on Definitions 3.12 and 3.13, we conclude that path equivalence falls into one of two categories: (1) equivalence with regards to the XML document and (2) equivalence with regards to the DTD/XML Schema. While the former is suitable for conventional XML query optimization, the latter constitutes the research here that uses semantics in XML Schemas to determine path equivalence.

3.3 Overview of Semantics & Features in XML Schemas

Semantics defined in XML Schema used throughout the thesis are outlined in this section. As we indicated in Figure 3.1, the semantics in XML Schemas are described by two groups of semantics including structural constraints and constraints of elements. The following sub-sections address the constraints in each group.

3.3.1 Structural Constraints in XML Schema

XML Schema provides a mechanism for constraining the document structure using *order*, *occurrence* of elements and *attributes*. In addition to these, the structural

constraints can also be derived based on how elements are defined within the schema.

In an XML Schema, a node can be an element which may be a complex type (denoted as *complexType*), and permits other elements to exist within its content and may also carry attributes. The complex type element is basically different from an element that is a simple type (denoted as *simpleType*) which cannot have an element content and cannot carry attributes. The element with a complex type and elements in its content have a *parent-child* relationship. That is, the element with a complex type is the parent of children in its content.

Figure 3.5 shows a portion of XML Schema, which depicts how the structural expressions formed by a sequence of elements **A** and **B**, **A** and **C**, **A** and **D**, **A** and **E**, **A** and **F**, **F** and **G**. The **A** element has a content of hierarchy elements **B**, **C**, **D**, **E** and **F** that make element **A** a complex type, that has a name **fullA**.

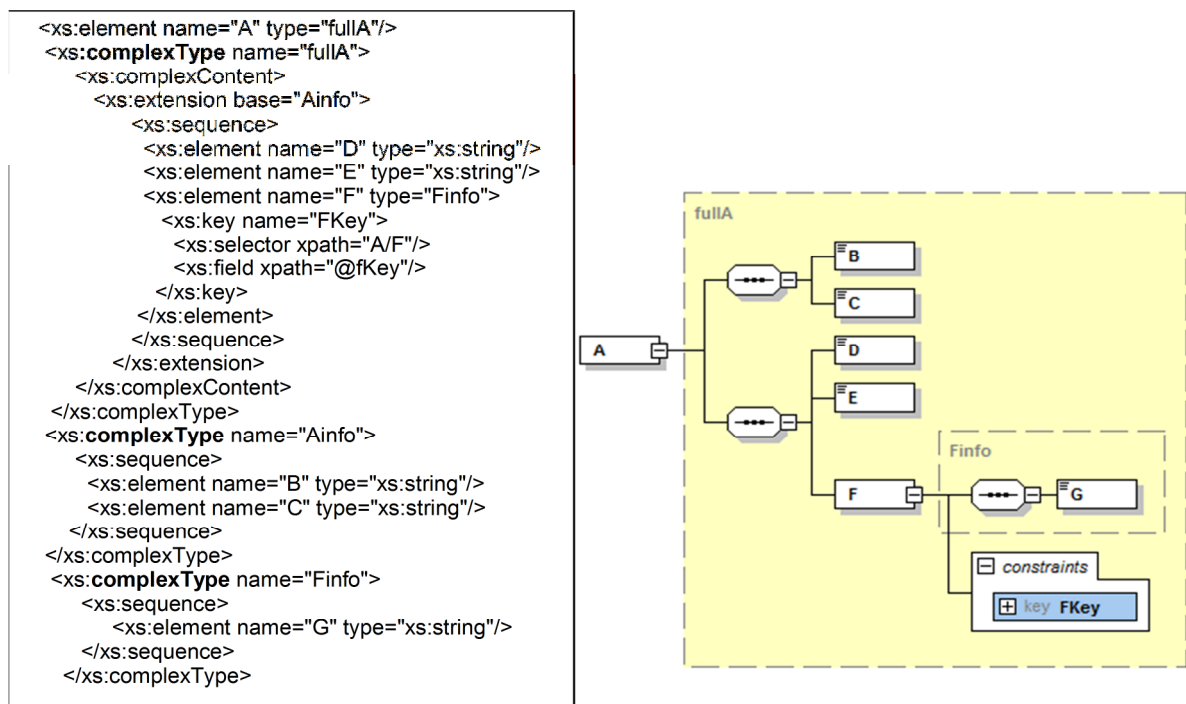


Figure 3.5 An Overview of XML Schema Specification

The complex type **fullA** shows a content that has a nested complex type which is made up by an independent complex type **Ainfo** and a number of elements (**D**, **E** and **F**). The complex type **Ainfo** contain a set of element (**B** and **C**), that are on the

same hierarchy with **D**, **E** and **F**. **Ainfo** demonstrates one of the powerful features in XML Schema that allows a reuse of **B** and **C** later somewhere in the XML Schema.

As element **F** appears as a type in the content of element **A**; hence, element **F** is a complex type, namely, **Finfo** has a content of one element **G** which demonstrates another level of hierarchy among the elements in the schema. We will explain the relationship between **A** and **G** after we show a hierarchical relationship between **A** and its immediate children **D**, **E**, **B**, **C** and **F**.

The *parent-child* relationship allows a pair of a sequence of elements to establish a path. Again in Figure 3.4, **A** and **B** can form a path using a *parent-child* relationship, which also applies to **A** and **C**, **A** and **D**, **A** and **E**, **A** and **F**. The next immediate hierarchical elements **B**, **C**, **D**, **F**, use a *parent-child* relationship to establish the further paths which can be more effective as hierarchical relationships have been skipped. Elements **B**, **C**, **D** do not have further hierarchical elements; however, element **F** has an immediate child: **G**, hence the existing path between **A** and **F** can be extended to **G**.

The *parent-child* relationship allows the complex element **A** and its content such as a set of children **B**, **C**, **D**, **E** and **F** to derive paths **A/B**, **A/C**, **A/D** and **A/E**, **A/F** and **A/F/G**. Elements in each path use a *parent-child* relationship, which is also referred to as a *structural constraint*.

A complex type element (denoted as *complexType*) has a content of a series of elements, in which some may be a complex type that has a content of another series of elements. The first element that has a complex type and the descendant of the children of the first element as well as its children, establish an *ancestor-descendant* relationship; that is, the element with a complex type is the ancestor, and the content in the lower hierarchies are the descendants (including the content of the first element).

The pair of elements **A** and **G** in Figure 3.5 can form a path **A//G** using an *ancestor-descendant* relationship *//*.

Both *parent-child* and *ancestor-descendant* relationships are categorized as structural constraints that can be derived from XML schemas.

In XML Schema, the *order* constraint is categorized as a structural constraint [W3C, 2004a]. The *order* constraint has a set of property values such as *sequence*, *choice*, *all* to allow the order and presence of the children within the parent element in the XML document.

The *All* property value enforces all the children of an element to appear in an instance in an un-restricted order, as specified in XML Schema.

The *sequence* property value enforces all the children of an element in an instance in the restricted order as specified in XML Schema.

The *choice* property value allow some children although may be not all of an element, to appear in an instance in the restricted order as specified in XML Schema.

Most XML Schemas use the *order* constraint with the *sequence* property value to control the order of the child elements within the parent element; this order constraint is very useful and important for semantic query transformation.

In Figure 3.5, an *order* constraint that has a *sequence* value is used in each complex type of element **A** and **F**. Children **B**, **C**, **D**, **E** and **F** of element **A** and child **G** in element **F** must appear in the XML document in an order as a set in XML Schema.

The *occurrence* constraint is also categorized as a structural constraint [W3C, 2004a]. It has a set of boundary values such as minimal and maximal occurrences denoted by *minOccurs* and *maxOccurs*. If the *minOccurs* of an element has a value 1 or greater than 1, then the element is required in XML documents. If the *minOccurs* of an element can have a 0 value, indicates the presence of the element is optional in a XML document. If the *maxOccurs* of an element has a positive value, it can be greater than or equal to the value of *minOccurs*, or it can be unbounded meaning it can be unlimited. When minimal and maximal occurrences of an element are not set, then the default is 1 for both of them.

In Figure 3.4, elements **A**, **B**, **C**, **D**, **E**, **F** and **G** do not have *occurrence* constraints set; therefore, the default occurrence for each element is 1 for both minimal and maximal values.

3.3.2 Constraints of Elements in XML Schema

XML Schema provides a large number of built-in data types and constraints for an element in XML Schema [W3C, 2004b].

Identity Constraint: XML Schemas can enforce a unique constraint using ID attribute and its associated attributes IDRef, IDRefs or Key and its associated attribute KeyRef. Note that Key and KeyRef are only available in XML Schema. These keys are categorized as *identity* constraints. The limitation of ID is that, since it is a type of attribute, it cannot be applied to attributes, elements or their contents, whereas Key and KeyRef can be created from combinations of elements and attribute content [W3C, 2004b].

For example, in Figure 3.6, the number attribute is **<field xpath="@number"/>** of those elements **<selector xpath="parts/part"/>** where reference (keyRef) name is **pKeyNum**, that has a valid location **regions/zip/part/partsNum** which in turn has a referenced value as **parts/part/@number**. This combination of an attribute and its content attribute cannot be achieved by using ID [W3C, 2004b].

```
<key name="pNumKey">
  <selector xpath="parts/part"/>
  <field xpath="@number"/>
</key>

<keyref name="pKeyNum" refer="pNumKey">
  <selector xpath="regions/zip/part"/>
  <field xpath="partsNum"/>
</keyref>
```

Figure 3.6 An Overview of an Identity Constraint using Key and KeyRef

There are many constraints that are useful to restrict an element with a simple data type. It is impossible to explain or describe all of them within the confines of the thesis. However, it is possible to describe those that are very commonly used in XML Schema. The following constraints are W3C standard constraints [W3C, 2004b].

Enumeration constraint. This constraint is used to define acceptance values by an element. For example, an element **State** has a string data type. The **State** element is restricted with a value such as State A, State B or State C. Any values other than these three values will be rejected when the element information is acquired. This kind of restriction is very useful for information processing when one knows that a certain type of information is available and a certain type of information is definitely not available in the repository.

Inclusive constraint. This constraint is used to restrict the upper and lower bounds of an element that has a numeric data type (e.g. integer). The lower bound (denoted by *minInclusive*), restricts the element that has a greater than or equivalent value. An upper bound (denoted by *maxInclusive*) restricts the element with a less than or equivalent value. For example, an element **zipcode** has an integer data type; this element is restricted with a value range between 1000 and 3000. If the **zipcode** is queried for any value beyond 3000 and below 1000, this will be treated as a conflict semantic query and as a result will be empty.

Exclusive constraint. This constraint is used to limit the exclusive lower and upper boundaries of an element that has a data type which is a numeric data type (e.g. integer). The lower bound (denoted by *minExclusive*) restricts the element that has a less than or equivalent value, and upper bound (denoted by *maxExclusive*) restricts the element with a greater than or equivalent value. For example, an element **age** has an integer data type, which is restricted with an exclusive value less than 65 and beyond 100. When **age** is queried for any values that are not between 65 and 100, the results are exclusive for **age**.

Pattern constraint. This constraint is used to define a certain pattern that elements can take on. For example, an element **password** has a string data type. The element is restricted with a pattern value `[a-zA-Z0-9]{8}`. Any password that does not have 8 characters including upper and lower values between a and z, A and Z, 0 and 9 would be considered invalid.

The *pattern* constraint is very useful to unify values and provide efficient spaces and resource utilization.

Length constraint. This constraint is used to specify the number of characters or list items allowed on an element. The length constraint also has proper value *minLength* and *maxLength*; these two properties allow more flexibility regarding the length within the boundaries. For example, an element **password** has a string data type. The element is restricted to a length of 8 characters with no restricted pattern. To be more flexible, the restriction of the length for a **password** element can also be set to a length between 8 and 16 using *minLength* and *maxLength*; this allows the user to specify a password of 8 to 16 characters.

Whitespace constraint. This constraint allows a whitespace (e.g. carriage return, line feed, tab, etc..) to appear in an element and can be handled accordingly.

For example, an element **address** has a string data type. The address element has a whitespace restriction and the property value:

collapse means that when there are white spaces such as carriage return, tab or line feed, they will be replaced with a single space.

replace means that when spaces are detected on the element, they will be removed.

preserve means that when spaces are detected on the element, they will be preserved.

3.4 Semantic Transformations – Summary of Problem Definition

The background information and semantic features described in Sections 3.2 and 3.3 will be used to determine problems in semantic query processing; these problems are then addressed by our semantic transformations.

Based on the semantics described in Section 3.3, the semantic transformation of an XML query into an equivalent semantic XML query (specific to XPath query) constitutes two important concerns. The first concern is the semantic transformation typologies that can provide a complete solution for transforming all types of XPath

queries. The second concern is to determine the optimization devices of the results produced by the first concern. The semantic transformation typologies to be addressed include:

- i. Semantic transformation typologies of *full path* expressions that consist of path notations such as $\{//, .., ., *, /\}$.
- ii. Semantic transformation typologies of *full path* expressions that support path notations such as $\{//, .., ., *, /\}$ as well as XPath axes such as *child*, *descendant-or-self*, *descendant*, *parent*, *self*, *ancestor*, *ancestor-or-self*, *preceding*, *following*, *preceding-sibling*, *following-sibling*.
- iii. Semantic transformation typologies of *partial path* expression. The partial path expression is accommodated with predicates where allowed conditions are varied and complex. Hence, the semantic transformation typologies will be broken down into an important structure of conditions.
- iv. Query condition with or without comparison value:
 - Single query condition with a comparison value, which is either a value or a path expression.
 - Multiple query conditions with disjunctive connections between the conditions
 - Multiple query conditions with conjunctive connections between the conditions
 - Multiple query conditions with both types of join conditions
- v. With path expressions and query component structures, questions can be asked to assist with the transformation starting with the following:
 - What are the most common expressions or query components that can represent the problem performance?
 - What are the semantics that can be identified so they can serve the transformations for certain types of XPath components?

- What is the basis used to identify the opportunities of transformation for optimization purposes?
- vi. The result of semantic transformation typologies will either produce a semantic XPath query or no semantic XPath queries. The semantic transformations focus on the following:
- To show the usefulness of semantics in XML Schema in query processing.
 - To modify the query structure in the presence of semantics, after being transformed.
 - To reduce the redundancies of query components to provide greater efficiency in resource utilization.
 - To provide a solution for query components that face database dependency challenges for more flexibility in information navigation.
 - To reduce the execution time of XPath queries after being transformed.
 - To identify semantic transformations as optimization devices. Not all semantic transformations guarantee an optimization. Therefore, the semantic transformation provides a systematic methodology where thorough experimentation can be carried out for optimization opportunities.
 - To provide a better adaptation for future extension of both semantic utilization and transformations.

3.5 Experiment & Performance Evaluation

After proposing semantic transformation typologies, the next step is to implement the proposed algorithms and carry out the evaluation process. The evaluation preparation process involves a few phases including experimental design and evaluation of query performance.

3.5.1 Experimental Design

Our experimental design section refers to a plan for conducting an experiment on a set of queries and data in order to obtain the results for an evaluation. The experimental design serves four main purposes:

- i. **A Background of Experiment Design:** this section addresses the objectives and evaluation of the strategy of experiments.
- ii. **Common Setup for the Experiments:** this section addresses the implementation framework and the database platform, keys to support minimal requirements for experiments, experimental data and schema selection, and setup overview and implementation modules.

The implementation framework and the database platform describe the working relationship between the semantic transformations and databases and how they enable the semantic transformations to work independently. The minimal requirement for experiments basically states some minimal resource requirements in order for the experimentation to be carried out.

The experiment data and schema selection describe the critical and central parts that guarantee the success of the experiment. The semantic transformations require semantics from XML Schema. Therefore, when choosing data for an experiment, we must take into consideration the semantics and structures in the XML Schema that conform to the data sets. To improve chances of opportunities to find optimization devices from the semantic transformations, two different sets of XML Schemas and data including real-life and synthetic ones are adopted. This is because real-life DBLP³ data is rich enough in semantics but structurally it faces a limitation of expressive hierarchies. Therefore, the conducting of complex queries may be challenging.

To overcome the challenges, we also adopt a Michigan benchmark data set as its data structure is very expressive [Runapongsa et al., 2006].

³ <http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/index.html>

Finally, the setup operational hardware, software and system modules section addresses the setup which includes software, hardware, data validation and loading.

For the database storage, a very well-known XML database management system that provides both XML native and XML-enabled storage management (XDBMS) has been selected by this research. For licensing purposes, the name of the vendor at this stage cannot be disclosed. The selection of XDBMS supports the native approach to store XML documents in RDBMS, and also facilitates the schema validation feature.

- iii. **Common Setup for the Experiments:** We mentioned that DBLP and Michigan XML Schemas and data are adopted. The structures and semantics in DBLP are different from those in Michigan Schemas and the data set. For each set of these data and schema, we carry out several improvement tasks such as semantic enhancement, data scaling, data cleansing, query taxonomy, metrics and computational procedures. Each of these tasks is addressed based on individual schema and datasets.

Semantic enhancement is carried out by studying the XML Schema both structurally and semantically to ensure they are well defined and that the semantics can be added. By doing this, we can create more opportunities for semantic transformations in XPath queries. Once the semantics in XML Schema have been modified, the data need to be revalidated in a data cleansing stage. The cleansing of data is based on semantic modification and scaling. When semantics of data sets are not sufficient adding more semantics means the contents of the data are also changed. Cleansing of data is very important as data not only needs to conform to the XML schema, but it also needs to be of high quality before being loaded into a repository.

With regard to a performance study, quality experimental data must be able to scale to several data sets, if we are to examine the query components that respond to various data sizes by applying the same type of semantic transformation. Therefore, scaling data in our experiment implies that different

result patterns may have an impact on the same query component for various data sets, under the same semantic transformation.

In order to apply the semantic transformations, the XPath query taxonomy needs to support both the semantics and components that are applicable to a semantic transformation typology. For a DBLP data set, the XPath query taxonomy is based on semantics available in the associated XML Schema. For each type of semantic transformation, we design a set of XPath queries that may return a sub-tree data or values of leaf nodes.

For the Michigan data set, the available benchmark provides XPath queries which are applicable to our semantic transformation. However, since not all benchmark XPath queries can be used, we select the XPath queries that demonstrate the expressive hierarchies and several types of conditions ranging from simple to a twig join [Wu et al., 2003].

As we adopt two sets of data including DBLP and the Michigan benchmark data, we propose the computational strategy that applies warm-up techniques during the execution stage where metrics such as transformation time and query execution are recorded for the transformed XPath query and its execution only. The computational strategy also applies special calculations for the recorded time as each XPath query is executed more than once.

If semantic conflict is detected in the XPath query, then the transformation will not produce any semantic XPath query.

3.5.2 Performance Evaluation

The proposed semantic transformations need to be evaluated thoroughly in order to identify individual transformation as optimization devices. For each XPath query and its semantic XPath query, the evaluation performance is based on the comparison between the recorded performance times (as described in the previous section) as well as the semantic rules applied to enable the success of the semantic transformation.

If the semantic XPath query is not produced due to semantic conflict in the original XPath query, the evaluation comparison is based on the transformation time and the execution time of the conflict XPath query against the execution time of the original XPath query.

3.6 Summary

We have provided an overview of the problems to be addressed by our proposed semantic transformations. The problem definition consists of three processes. The first uses schema semantics to transform XPath queries with specified simple path operators. The second uses schema semantics to transform XPath queries with specified XPath axes. And the third uses schema semantics to transform XPath queries specified with predicates.

This chapter has also provided the background of XML and query essentials. The chapter has presented the basis related to XML documents, Schemas and data models such as components and terminologies related to XML query structure and processing. It has then addressed the notions of XML query processing.

The chapter has also provided an overview of XML semantics and properties; it discusses the types of semantics including structural semantics and the semantics of elements available in XML Schemas. Problems are then summarized based on the problem definitions and available semantics as provided for proposed semantic transformations. Finally, the experimental design and performance evaluation methodology were discussed in detail.

In Chapter 4, a pre-processing semantic methodology is presented that is proposed for pre-processing schemas semantics and efficiently storing them for the proposed transformation typologies as required. Following this, the semantic transformation typologies for the first group of problem definition will be proposed.

Chapter 4

Derivation of Semantics & Semantic Path Transformation

The main objective of this research is to utilize semantics in XML schemas for the purpose of semantic query transformations. In order to achieve the objective, semantics need to be derived from given XML Schemas to support the transformations proposed. This chapter proposes a framework for deriving semantics from XML Schemas and semantic path transformations.

4.1 Derivation of Semantics

Since XML Schema is very rich in semantics, deriving semantics during the transformation process is inefficient in terms of both time and resource consumption. Such an approach will cause deterioration in performance. Without the semantic derivation prior to the utilization of semantic transformations, the required semantic will need to be searched throughout the whole schema and further processed if found. We believe that the derivation of semantics not only expedites the transformation process, but also makes it easier to retrieve semantics when required.

XML Schema contains two types of constraints: *structural constraints* and *constraints of elements* [Paparizos et al., 2007; W3C, 2004a; 2004b].

A *structural* constraint consists of path constraints [Fan, 2005] that are unique to XML Schemas. The first type of a path constraint is the parent-child relationship of two *sequence* elements along an edge of a schema tree. The second type of path constraint is the ancestor-descendant relationship [Che et al., 2006] of two elements that occurs along an edge of a schema tree. The third type of path constraint is the identity constraint, which provides uniqueness or reference with respect to multiple elements or attributes. The content of identity constraint is a path that combines the relationships of both parent-child and ancestor-descendant [W3C, 2004a].

We use the parent-child constraint to build unique paths. Instead of numbering the elements in the path [Kha & Yoshikawa, 2004], we use the element tag names in the path and make the paths unique.

Definition 4.1. (Unique Path): A unique path q derived from a given schema S is a sequence of elements $E = \{e_1, e_2, e_3, \dots, e_n\}$ that traverses from e_1 to e_i along an edge where $1 < i \leq n$. There exists only parent-child “/” relationship among the elements e_i in q .

Two types of elements can be found in a XML Schema. The first one is a complex type (specified as *complexType* in XML Schema) element, which defines an XML element that contains other elements and/or attributes. The second one is the simple type (specified as *simpleType* in the XML Schema) which defines a schema element that is an atomic or built-in data type [W3C, 2004a].

Figure 4.1 shows that an **employee** element has a **fullpersoninfo** complex type, which is created by extending the existing complex type **personinfo** (using an extension base) and three additional elements of **address**, **city** and **country**, which are simple types. The **personinfo** complex type contains **firstname**, **lastname** and **age** elements which are referred to as simple types because they have atomic data types and do not contain other elements.


```

<xs:element name="employee" type="fullpersoninfo"/>
<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
    <xs:element name="age">
      <xs:simpleType>
        <xs:restriction base="xs:integer">
          <xs:minInclusive value="18"/>
          <xs:maxInclusive value="99"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="fullpersoninfo">
  <xs:complexContent>
    <xs:extension base="personinfo">
      <xs:sequence>
        <xs:element name="address" type="xs:string"/>
        <xs:element name="city" type="xs:string"/>
        <xs:element name="country" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

```

Figure 4.1 ComplexType and SimpleType Elements

A parent-child relationship (also known as a structural constraint - as described in Chapter 3), exists in the schemas between **employee** and **firstname**, **employee** and **lastname**, **employee** and **age**, **employee** and **address**, **employee** and **city**, and **employee** and **country**. The parent-child relationship is important and critical for our unique path derivation.

The second type of a constraint in XML Schema is the constraints of elements. Each schema element has at least one constraint. The compulsory constraint is a modelling constraint [Ferrarotti et al., 2011; Link & Trinh, 2007] known as an *occurrence* [W3C, 2004a]. The *occurrence* constraint states the cardinality of children present in the sub-tree as it is useful in most of our proposed semantic transformations. Its usefulness will be seen later in our transformations in Section 4.2, Chapters 5 and 6. Apart from the *occurrence* constraint, a schema element may have other semantics depending on its type.

When the schema element is a complex type, compulsory *occurrence* and *order* constraints are implemented. The *order* constraint has a set of values {sequence, choice, all, group}, which are used to restrict the correspondence of the elements that are contained within the complex type element. Figure 4.1 shows the *sequence* value of *order* constraint used by the **personinfo** and **fullpersoninfo** complex types to restrict the order of their elements. For **personinfo** complex type, the order of its elements is **firstname**, **lastname** then **age**.

As **personinfo** complex type has a sequence value for *order* constraint, its content is in the ordered elements including **firstname**, **lastname**, **age**. Hence, the order of elements in **fullpersoninfo** complex type is **firstname**, **lastname**, **age**, **address**, **city** and **country**.

An element may be a simple type. For example, an element **age** has an *occurrence* constraint with a minimal occurrence of 1 and maximal occurrence of 1. In addition to this, an **age** element will have an *integer* type and is set with an *inclusive* constraint that has a value between 18 and 99; this shows that the **age** element is a simple type that has two constraints: *occurrence* and *inclusive*.

Figure 4.2 depicts a set of common XML Schema constraints. The downward arrows indicate they are open for more constraints to be added to the list. An element can be identified as an Attribute, a complexType or a simpleType. The difference between the complexType and simpleType elements is that a complexType can carry an attribute and/or element content, which can be simpleType or another complexType; for example, the complex type **fullpersoninfo** carries **personinfo** complex type in Figure 4.1, or both. The simpleType element does not carry an attribute and or element content. SimpleType element and attribute are atomic data types. However, simpleType element is where data is directly specified [W3C, 2004a; W3C, 2004b]. Therefore, a simpleType element is one that is very rich in semantics.

Even though each type of element has its own associated semantics, there are two constraints worth mentioning. The *occurrence* constraint is shared by an element that is either a complex type or simple type. The *identity* constraint is shared by an element that is an attribute or a simple type whose content is referenced by another element. The constraints in Figure 4.2 are W3C standard recommendation. Details of

each constraint can be found in [W3C, 2004a; 2004b]. All the constraints in Figure 4.2 are derived from the list of constraints of elements (details of which are provided in Section 4.1.2).

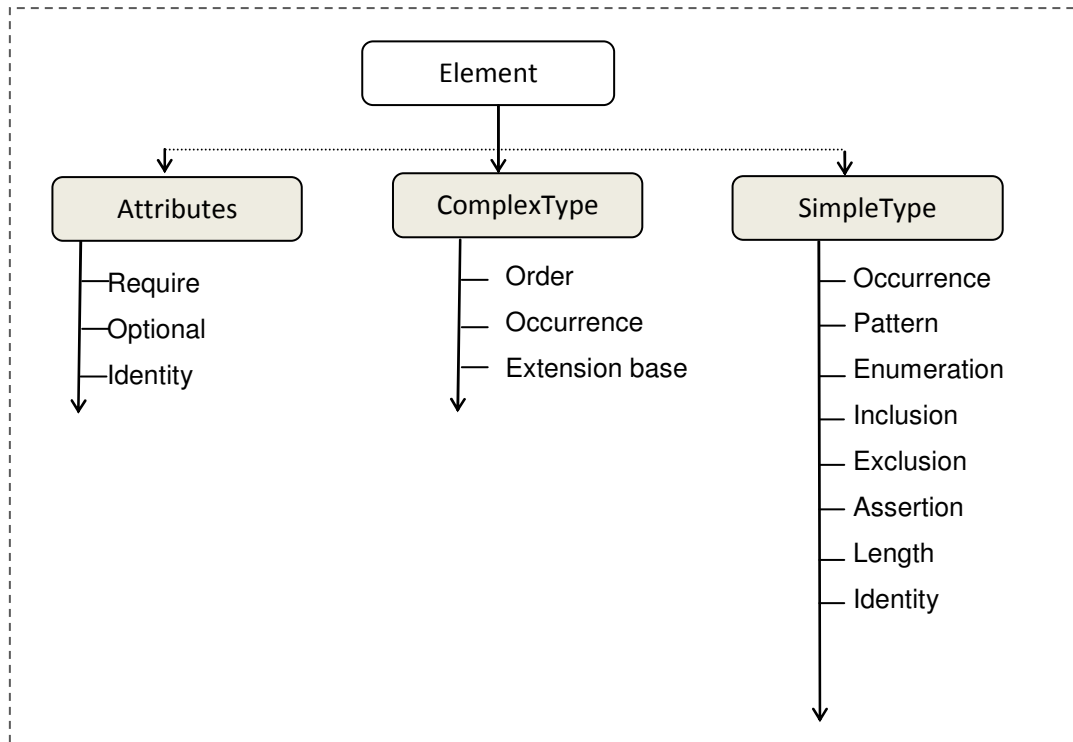


Figure 4.2 Some Common Constraints in XML Schema

This research proposes methodologies for semantic derivation. The first methodology is to derive unique paths. The second methodology is to derive constraints of each individual element.

4.1.1 Deriving Unique Paths

We derive the unique paths by adopting the depth-first search approach [Thomas et al., 2001; Jin et al., 2011].

Definition 4.2. (Depth First Search (DFS)): a depth-first search is a searching approach that explores the nodes along the edges of a directed tree, i.e. XML schema as far as possible first, before backtracking.

Figure 4.3 illustrates how the DFS is applied to derive a set of unique paths for all nodes on a tree structure.

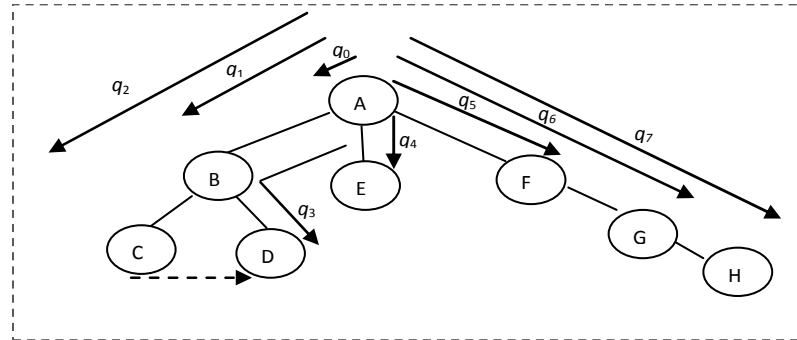


Figure 4.3 Unique Path Derivation using Depth-First Search

As shown in Figure 4.3, let Q be a list that contains a set of unique paths. Each unique path is derived based on Definitions 4.1 and 4.2.

$$Q = [q_0, q_1, q_2, q_3, q_4, q_5, \dots, q_n] \text{ where } n > 0$$

The directional numbering arrows (excluding the dash-arrows) represent a set of unique paths $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\} = \{A, A/B, A/B/C, A/B/D, A/E, A/F, A/F/G, A/F/G/H\}$ in order of sequence. In Definition 4.1, we define that a unique path must be expressed from the root to any element in the schema along the path; this simply means that all unique paths must begin with a root element.

The derivation of unique paths greatly influences the semantic transformation for the XPath axes such as *following*, *preceding*, *following-sibling* or *preceding-sibling*. The order of unique paths helps speed up the search of the elements; this is important since semantic transformations for most XPath axes require the order of the elements to accomplish the tasks.

For example, based on Figure 4.3, we want to retrieve all the following members of element D under B.

The following members occur after D in the depth-first search order of the nodes. To achieve this, the *following* axis is designed for this purpose. In semantic transformation with the ordered unique paths, the transformation could not have been easier.

Let us consider Figure 4.3 where an XPath query for the above requirement is:

$$p = A/B/D/following::*$$

From the list of unique path list Q , we can locate $\{A/B/D, A/E, A/F, A/F/G, A/F/G/H\}$ because these unique paths come after $A/B/D$ in Q . These unique paths are schema paths; therefore, the final result that responds to the requirement would be the set of unique paths $\{A/B/D, A/E, A/F, A/F/G, A/F/G/H\}$ excluding the first occurrence of element D . This can be easily handled by using the context position function $position()$, which will be addressed later in the semantic transformation [W3C, 1999; 2007a; 2010].

The significance of deriving unique paths in an ordered manner is to eliminate the unnecessary searching and matching of ordered elements when dealing with the transformation of XPath axes.

This section has proposed a methodology for deriving unique paths and ordering them, based on the order constraint of the elements. In the next section, we propose a methodology for deriving the constraints of elements.

4.1.2 Deriving Constraints of Elements

Constraints of elements in XML Schemas are very rich. An element may have a number of constraints assigned to it.

We propose a methodology for deriving the constraints and their associated values of an element as follows.

- W is a set of constraints in a given schema S where $W = \{w_1, w_2, \dots, w_n\}$ and V is a set of values where $v_i = \{v_{i,1}, v_{i,2}, v_{i,3}, \dots, v_{i,l}\}$ for a constraint w_i , $1 \leq i \leq n$ and $1 \leq k \leq l$
- e_j is an element that has constraint w_i where j is a sequence that starts with 1.
- C is a list where $C = [c_1, c_2, c_3, c_4, c_5, \dots, c_y]$, $1 \leq x \leq y$
 c_x is composed as $c_x = z/e_j w_i V$
 z is a path formed by an element or a sequence of elements e separated by '/'.
 $z/e_j = e_0/e_1/e_2/\dots/e_j$ where $1 \leq j$.

Based on Figure 4.3 let us consider elements B and E, each of which has an *occurrence* constraint with a minimal and maximal occurrence of 1. Element D has a minimal occurrence of 1 and maximal occurrence of 5. B also has an *inclusive* constraint with inclusive range values between 1 and 20. The derivation of list *C* is shown below:

$C = [A/B \text{ occurrence } 1 \ 1, A/B \text{ inclusive } 1 \ 20, A/E \text{ occurrence } 1 \ 1, A/B/D \text{ occurrence } 1 \ 5, \text{etc...}]$

From the derived *C* we can interpret $c_1 = A/B \text{ occurrence } 1 \ 1$ where the value of *z* is A, the value of *e* is B, the value of *w* is occurrence and the value of *V* is 1 1.

We may now state that *C* is a list that contains elements, constraints of associated elements and values of associated constraints of elements.

By deriving the semantics of each element in the proposed manner, items in the *C* list can avoid duplication issues.

This section has proposed a derivation of semantics methodology for constraints of each schema element to create a *C* list. From this point onwards, our semantic transformation rules will utilize the information in *Q* and *C*. Both *Q* and *C* are treated as global inputs to semantic transformations.

4.2 Semantic Path Transformation

This section proposes the semantic path transformations. The inter-relationships of semantic path transformations are presented in Figure 4.4.

We present a set of semantic transformation typologies to transform a simple XPath query with fragment $\{/, //, *, \dots, ..\}$ into equivalent semantic XPath queries by using unique path list *Q* and constraints of element list *C*.

Unique paths represent structural constraints as elements in each unique path and are connected by a parent-child '/' relationship. The constraints of elements in list *C* also play an important role in assisting with semantic path transformations. We show how

performance changes significantly when such constraints are applied in semantic path transformations.

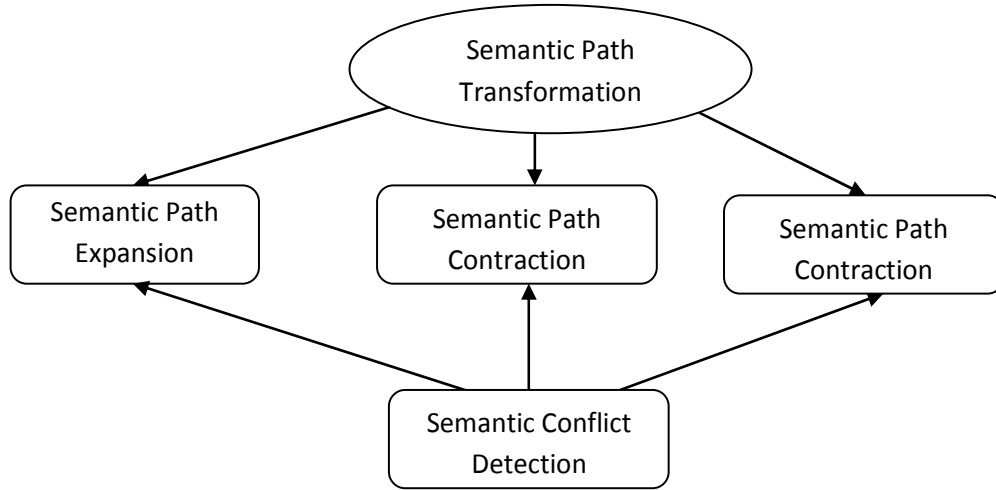


Figure 4.4 Inter-relationships of Semantic Path Transformations

4.2.1 Semantic Path Expansion

This section proposes a technique to transform an XPath query that has a path fragment ‘//’ ancestor-descendant relationship between two location steps (refer to Definition 3.6 in Chapter 3).

Definition 4.3. (Semantic Path Expansion - SPE) Semantic path expansion is a transformation that replaces an ancestor-descendant ‘//’ in the XPath query with a path fragment of a sequence of elements that has only a parent-child ‘/’ relationship among a set of sequence of elements.

Following Definition 4.3, we can develop SPE rules as a guideline to produce the semantic XPath query. First, let us consider the following parameters.

Let:

- p be the given XPath query
- β be a fragment represented by // in p
- S (abbreviation of **semanticXPath** list) be the list of transformation results

SPE Rules. The semantic path expansion of β occurs when all elements in the XPath query are successfully matched to elements in a unique path $q \in Q$ and $'//'$ can match to a fragment in q . Semantic XPath query S is produced such that $S = \{q\}$ if and only if the number of matched q is 1 and it must also satisfy one of the following rules:

1. a target element in matched q is a descendant element in β where the descendant element must be a leaf element;
2. a non-target element (refer to Definition 3.8 in Chapter 3) in matched q and p is also a descendant element in β where the descendant element is a non-leaf element.

The SPE rule (1) simply means that if there is a fragment $'//'$ in the XPath query where the descendant in p is in the left most location step, this descendant element must appear in identified q as a target element, which must be a leaf element in the schema.

Given XPath query $p = */b//d/f$ and $Q = \{q_1, \dots, q_9\}$ as shown in Figure 4.5.

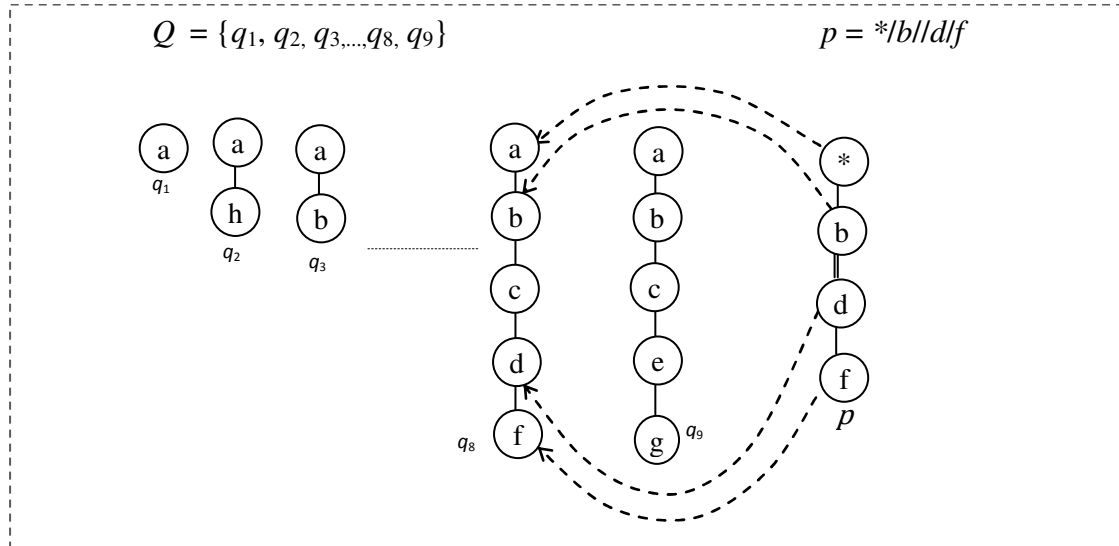


Figure 4.5 XPathQuery p and List of Unique Paths Q

Based on the structure and locations of β where $\beta = b//d$, the SPE rule (2) is identified as being the suitable one. It shows the descendant element d in p is a non-target element in both p and q . The matching process adopts a bottom-up approach. It first matches all valid elements in p to each element in each $q \in Q$ (q_1, \dots, q_9 in Figure

4.5). It then matches $//$ by using the path fragment $\mathbf{b//d}$ to find the unmatched path fragment in q , which is enclosed by elements \mathbf{b} and \mathbf{d} . To match $*$, it finds the unmatched element in q that is the parent of element \mathbf{b} . β is now matched to the fragment $\mathbf{b/c/d}$ in q_8 and $*$ can be mapped to the parent of element \mathbf{b} , which is element \mathbf{a} . As the result, it produces semantic XPath query $S = \{q_8\}$. Let us consider an example based on the DBLP schema (which is presented in **Appendix1**). Q and C are derived from the DBLP schema using a semantics derivation technique.

Requirement. XPath query selects the title names of all theses in the DBLP database.

XPath query $p = */phdthesis//tn$

Value of $\beta = phdthesis//tn$

From the schema:

$$Q = \{ \text{dblp}, \text{dblp/article}, \text{dblp/article/author}, \dots, \text{dblp/phdthesis}, \\ \text{dblp/phdthesis/author}, \text{dblp/phdthesis/title}, \text{dblp/phdthesis/title/tn}, \dots \}$$

Now we apply the semantic transformation SPE rules. We use a bottom-up matching approach by which both elements **phdthesis** and **tn** elements can be matched to q_i where $q_i = \text{dblp/phdthesis/title/tn}$ in Q . As **tn** is in the right-most location step of p , **tn** is a target element in both p and q_i .

Once the q_i has been identified, the transformation uses fragment **phdthesis/title/tn** in q_i to replace the fragment **phdthesis//tn** in p and the fragment **dblp/** in q_i replaces fragment $*/$ in p . The semantic path expansion transformation satisfies the SPE rule (1) where it produces a semantic path query $S = \{q\} = \text{dblp/phdthesis/title/tn}$

We now propose Function 1, namely **semanticPathExpansion**, to implement the proposed semantic path expansion rule.

Function 1 is called upon when $“//”$ is detected in a given XPath query. Input parameters semanticXPath and p are prepared and passed in by the main algorithm, prior to calling Function 1. Note that Q and C are the list of unique paths and the list of constraints of elements respectively.

The descendant element in p is first determined as either a target element or a non-target element (Lines 1:1-1:4) so that verification can be carried out appropriately. For each unique path in `tempList`, the transformation matches information accordingly and obtains q that meets the required information. The matched q is then stored in the `semanticXPath` list. During the matching process, the transformation takes care of the case where both descendant and ancestor elements are specified with valid element tag names or '*' or 'node()' (Lines 1:3-1:4).

When the descendant element in p is detected as a non-target element (Lines 1:5-1:9) in XPath query p , the transformation also takes care of the case when both descendant and ancestor elements are specified with valid element tag names, or specified with '*' or 'node()'.

Function 1: List semanticPathExpansion(List semanticXPath, String p)

```

    Let  $\beta$  be fragment in  $p$  where '/' exists,  $\partial \in \beta$  be descendant element,  $\theta \in \beta$  be ancestor element,
    tempList be empty lists, target=false be Boolean, o={*, node()}, Q be the list of unique path q
1:1  For each q in Q
1:2    If found  $\partial$  and  $\theta$  in q and  $\partial$  is not a complex element type Then
1:3      push q to semanticXPath
1:4  End Loop
1:5  If target == false Then
1:6    For each q in Q
1:7      If (( $\partial$  or  $\theta$  or both found as non-target element in q)) (  $\partial$  and  $\theta$  are '*' or node()) Then
1:8        push q to tempList
1:9    End Loop
1:10 If tempList is NULL Then semanticXPath = NULL
1:11 Else if tempList is not NULL Then semanticXPath = tempList
1:12 If semanticXPath is not NULL Then
1:13   For each s in semanticXPath
1:14     verify all elements in p to current s
1:15     If (( $\partial$  and  $\theta \in O$ ) && target = true ) Then
1:16        $\partial$  must not contain further descendants
1:17   End Loop
1:18 If length(semanticXPath) > 1 Then semanticPath = p
1:19 Else semanticXPath = 'Retain'
1:20 Return semanticXPath

```

When both the descendant and the ancestor are '*' or node(), the transformation must first ensure that all the elements in p are matched with the descendant element. It then verifies the descendant as a target element and confirms that the target element

in matched q is a leaf node; otherwise, the result would be inaccurate (Lines 1:12-1:17).

The semanticXPath is produced only if the final result holds only one semantic XPath query. If multiple unique paths are produced for semanticXPath, a message ‘**retain**’ is assigned to semanticXPath to indicate the main algorithm; that is to say, the XPath query is valid but cannot be expanded (Lines 1:18-1:19). Finally, the semanticXPath is returned to the main function (Line1:20).

When the semanticXPath list is NULL (Line 1:10), structurally the XPath query does not satisfy any q in Q . This requires the application of a semantic conflict detection rule proposed in Section 4.2.4. The function returns NULL since semantic conflict has been detected.

4.2.2 Semantic Path Contraction

This section proposes a semantic path contraction (SPCon) transformation to transform an XPath query specified with a wildcard ‘*’ to a single semantic XPath query without a wildcard ‘*’. The wildcard ‘*’ in this case represents multiple path fragments [Wu et al., 2008].

When ‘*’ represents a single element in the XPath query, the transformation produces a single semantic XPath query, to which a SPE is applied. Otherwise, SPCon is responsible for the semantic XPath query when wildcard ‘*’ represents a set of elements or path fragments. SPCon produces multiple unique paths which are then contracted to a single semantic XPath query by using ‘//’ to replace the different path fragments in the multiple unique paths.

Definition 4.4. (Semantic Path Contraction – SPCon) Semantic path contraction is a transformation to replace ‘*’ in the user XPath query with ‘//’ in which ‘*’ represents a set of path fragments in the multiple unique paths.

Based on Definition 4.4, we develop a SPCon transformation rule. First let us consider the following parameters.

Let:

- p be the given XPath query
- β be a fragment in p where β holds '*', e.g. '*/*', 'a/*', '*/b'
- S (abbreviation of **semanticXPath** list) be the list of transformation results

SPCon Rule. The SPCon transformation proceeds when β is detected in the XPath query p . '*' is successfully matched to elements that are located in $\{q_i, \dots, q_j\}$ where $q \in Q$, if one of the following rules is satisfied:

1. given $i \neq j$ that q_i to q_j share the same target element, which must match the target element in p there exists some different path fragments among q_i to q_j or;
2. if $i = j$ then $S = \{q\}$ and no contraction occurs, as only one semantic XPath is produced or;
3. if $i \neq j$ $S_{(\text{SPCon})} = \{q_i, \dots, q_j\} = \{q\}$ where q constrains // that represents different path fragments among $\{q_i, \dots, q_j\}$ except the target element.

The semantic rule focuses on the target element in all the identified unique paths which must have the same target element. For example, if p is a path $a*/c$, all the identified unique paths are $a/b/c$ and $a/d/f/c$ in which both traverse in two different paths but start from the same root a and reach the same target element c .

If * is specified as the right-most element in p , there is a possibility that * represents the different fragments or elements, in which case the SPCon rule is not applicable.

In Figure 4.6 we demonstrate the XPath query p (right most query tree), which can be contracted based on unique paths in Q (left most query trees). As the mapping process (a bottom-up approach) starts from the target element g in p to the same target element g in each unique path q , unique paths q_{10} , q_{15} and q_{16} are identified.

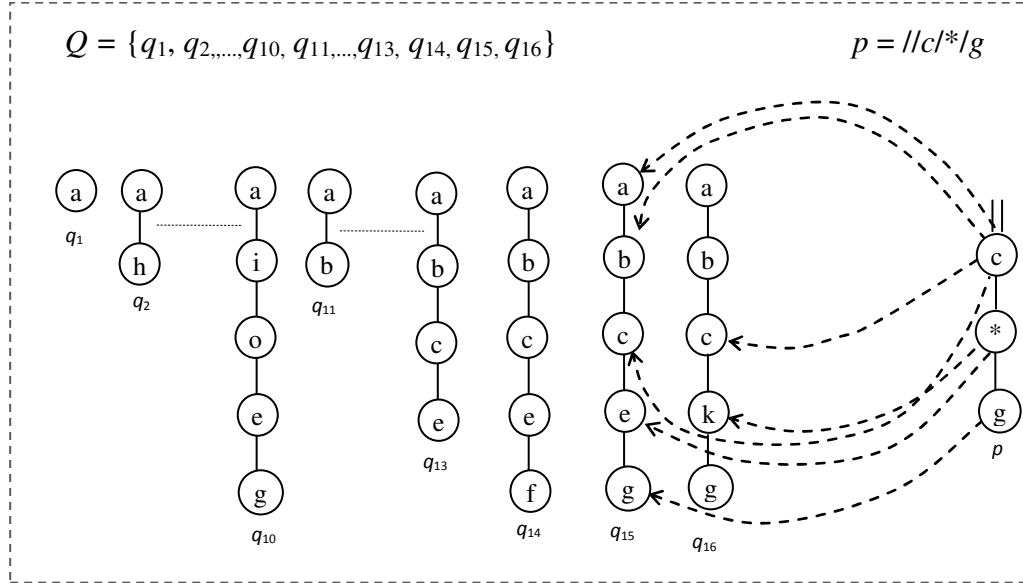


Figure 4.6 XPath Query p and list of Unique Paths Q

When matching from element c in p to elements in q_{10} , q_{15} and q_{16} , the SPCon transformation drops q_{10} , which does not contain element c . In p , the SPCon solves the problem of wildcard $*$ in path fragment $c/*/g$. Given unique paths q_{15} and q_{16} , two fragments $c/e/g$ in q_{15} and $c/k/g$ in q_{16} are identified as corresponding to fragment $c/*/g$ in p .

The SPCon transformation contracts $\{q_{15}, q_{16}\}$ that produces $S = //c//g$

Let us consider an example based on the DBLP schema (which is presented in **Appendix 1**). Q and C are derived from the DBLP schema using a semantics derivation technique proposed in Section 4.1.

Requirement: XPath query to select all the title names of all existing items (such as ‘article’, ‘proceedings’, ‘inproceedings’, etc...), in the DBLP database.

XPath query $p = \text{dblp}/*/title/tn$

In the given p there exists $\beta = \text{dblp}/*/title$

In p , element tn is the target element in some unique paths q where $q \in Q$. As the mapping process (a bottom-up approach) starts from target element tn in p to the same target element tn in any q , we find more than one unique path q that satisfies the fragment represented by $*$ in p .

$Q = \{ \text{'dblp/article/title/tn'}, \text{'dblp/inproceedings/title/tn'}, \text{'dblp/proceedings/title/tn'}, \text{'dblp/book/title/tn'}, \text{'dblp/phdthesis/title/tn'}, \text{'dblp/incollections/title/tn'}, \text{'dblp/www/ title/tn'} \}.$

The SPC_{on} rule produces the semantic contracted path $S = \text{dblp//title/tn}$. The contraction occurs at the fragments that contain different elements such as **article**, **inproceedings** ..., **www** in all located q .

Function 2 (below), **semanticPathContraction**, accepts two inputs which are semanticXPath list and XPath query p from the main algorithm. The function is designed based on the proposed SPC_{on} semantic rule; that is, $*$ is allowed to appear in any XPath location step.

Function 2: List semanticPathContraction (List semanticXPath, String p)

Let β be fragment $*$ or node() in p , Q be the list of unique path q , d_q be different path fragments

```

2:1 For each  $q$  in  $Q$ 
2:2   If elements in  $p$  match elements in  $q$  Then
2:3      $\beta$  be a path fragment contained all un-matched elements in  $q$ 
2:4     Push  $q$  to semanticXPath
2:5   End Loop
2:6 If Length of semanticXPath greater 1 Then
2:7   Locate  $d_q$  in all  $q$  in semanticXPath
2:8   Replace all  $d_q$  with  $//$  in  $q$  that ultimately derive a contracted  $q$ 
2:9 Else length of semanticXPath is 0 Then semanticXPath =  $p$ 
2:10 Return semanticXPath

```

The **semanticPathContraction** contracts the path fragments labelled with $*$ or node() in p . The contraction contracts only the fragment of $*$ or node() that is a non-target element. It first checks to ensure that the valid elements in p match elements in q . It detects $*$ or node() in q and pushes q to the semanticXPath list. During this stage, a semantic conflict may be detected (Lines 2:3). Otherwise, it produces a semanticXPath list (Lines 2:1 – 2:5).

If the semanticXPath list is produced, that means semantic XPath queries are identified to produce the result set. Contraction is performed to contract the path fragments that are different in the multiple semantic XPath queries. If semanticXPath is NULL, this means that a conflict has been detected in (Line 2:9). If there is more

than one semantic path in the semanticXPath list, then contraction proceeds in order to produce a single path (Lines 2:6 – 2:8). It finally returns the semanticXPath list to the main algorithm (Line 2:10).

When the semanticXPath is NULL it means that either the elements in p or the structure in p do not match information in Q . If the semanticXPath returns empty to the main function, it determines the conflicts detected in p (Definition 4.6).

4.2.3 Semantic Path Complement

A semantic path complement transforms a given XPath query specified with a parent location step ‘..’ to a semantic XPath query without the parent ‘..’ operator.

Definition 4.5. (Semantic Path Complement - SPC_{om}) A semantic path complement is a transformation which transforms operator ‘..’ or the parent axis location step by eliminating ‘..’ and the element that occurs before it.

We now develop SPC_{om} rules as a guideline to achieve the transformation goal.

Let:

- p be the given XPath query
- δ be an element that occurs in a location step next to and before the location step containing only ‘..’
- ϑ be an element that occurs in a location step next to and after the location step containing ‘..’
- θ be an parent element of δ and ϑ
- β be the fragment in p that is formed as $\delta/./\vartheta$, ϑ and δ are in β
- S (abbreviation of **semanticXPath** list) be the list of transformation results

SPC_{om} Rule. The SPC_{om} proceeds when there exists β in which δ is * or ϑ is * or both δ and ϑ are *. The path fragment in p leading * and/or ‘..’ matches a fragment in unique path q_i and the path fragment in p tailing ‘..’ and/or * matches a fragment in

unique path q_j . That is, q_i and q_j share the same ancestors of elements ϑ and δ . Semantic XPath query $S_{(\text{SPCom})} = \{q_j\}$ if and only if

1. Target element ϑ in q_i has a minimal occurrence (retrieved from list C) greater than 0
2. There exists a branching element θ in q_i and q_j
 $\therefore i \neq j$ and some ancestors of δ are also ancestors of ϑ .

The **SPCom** rule imposes a structure on p that matches two unique paths q_i and q_j in Q where q_i and q_j have ϑ as a target element. ϑ has a maximal occurrence (verified in list C) of at least 1, ϑ always exists in the database regardless of whether or not δ exists.

The SPCom not only eliminates the unnecessary use of the operator ‘..’ in the XPath query p , but also makes use of SPConn and SPE transformations to achieve the best possible optimization.

Figure 4.7 demonstrates the XPath query p (left most query tree), which can be complemented based on unique path information provided in Q (right-most query trees where bold nodes are the branching nodes).

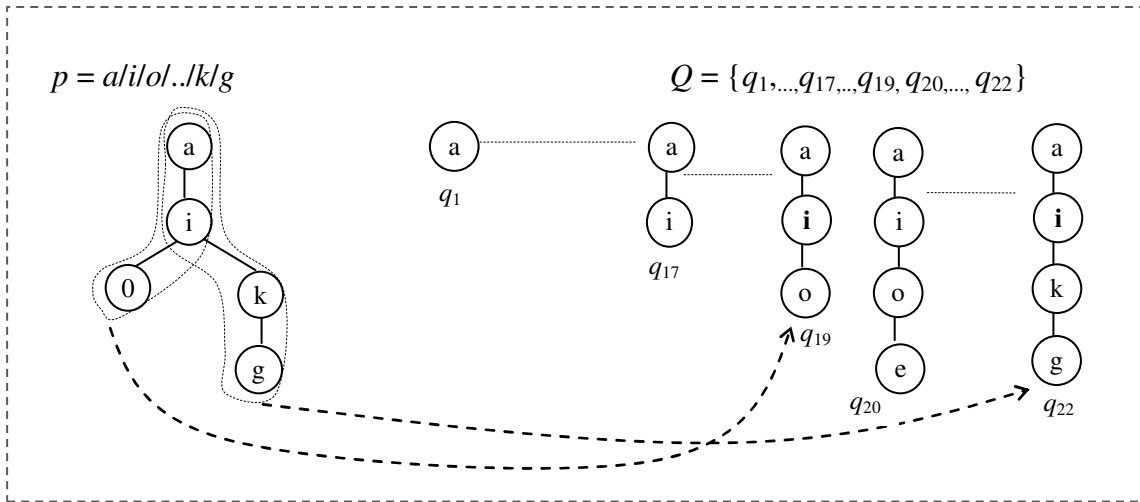


Figure 4.7 XPath Query p and List of Unique Paths Q

Due to the presence of ‘..’ in p , the query tree shows a branching element in p . This indicates a number of required unique paths based on the number of branches, after the branching node **i**.

The mapping process (a bottom-up approach) starts from the target element g in p , traversing along the edge to the root (this path is referred to as the right branching path) a in p , to the same target element g in each unique path q . Hence, the unique path q_{22} is identified.

The process repeats for the left branching path, which starts from the o element traversing along the edge back to root a in p ; hence, we identify unique path q_{19} .

By following semantic rule SPC_{om} , two unique paths $\{q_{19}, q_{22}\}$ have been identified. If the occurrence constraint defined for element o is $(1, \infty)$, then we can use rule SPC_{om} to remove fragment $o/./$ from the XPath query p . The ancestors of element o are also the ancestors of g . S is now produced as $S = ali/k/g$

Let us consider an example based on the DBLP schema and a list of derived unique paths Q and a list of constraints of elements C .

Requirement: XPath query to select all titles of theses that must have valid authors in the DBLP database.

XPath query $p = dblp/phdthesis/author/./title$

In the given p there exists $\beta = \mathbf{author/./title}$, $\delta = \mathbf{author}$, $\vartheta = \mathbf{title}$

In XPath query p , target element **title** is first matched to a target element in any $q \in Q$, which initially produces potential q such that $\{q_i, \dots, q_k\} = \{dblp/article/title, \dots, dblp/phdthesis/title, \dots, dblp/www/title\}$. As a result, more than one unique path has **title** as a target element.

From the potential $Q = \{q_i, \dots, q_k\}$, it then matches the elements along the edge of element **title** in p which are **dblp** and **phdthesis**. These elements match elements in one of the potential unique paths such as **dblp/phdthesis/title**. So now the valid path for selecting information is identified as **dblp/phdthesis/title**.

Next, the transformation identifies the condition element **author** to any unique path q in Q that has **author** as the target element, which produces another set of potential $\{q_n, \dots, q_z\} = \{dblp/article/author, \dots, dblp/proceedings/author, dblp/phdthesis/author, \dots\}$

The elements **dblp** and **phdthesis** occurring along the edge of **author** are then matched to one of q in potential list $\{q_n, \dots, q_z\}$ to find the matched q for elements **dblp**, **phdthesis** and **author**. The matched q is **dblp/phdthesis/author**.

Finally, the transformation needs to verify the occurrence constraint of element **author** under parent **phdthesis** in list C , which indicates an entry of ‘**phdthesis/author occurrence 1 ∞** ’ presented. This means that for every **phdthesis** in **dblp**, there must be at least one associated **author**. It now removes fragment **/author/..** from p , which produces:

$$S = \text{dblp/phdthesis/title}$$

We now propose Function 3, namely **semanticPathComplement** to achieve the semantic rule SPC_{om} .

Function 3 accepts two inputs. The first one is semantic XPath query list **semanticXPath** and the second one is the XPath query p . In this transformation, we consider **semanticXPath** list to be empty. The transformation always uses a unique path list Q .

Function 3 first separates p into two paths, p_1 and p_2 , based on information such as ϑ , $\hat{\partial}$ and θ . The transformation starts with p_1 by matching all possible information regarding elements and structure to q in Q . The matching process considers several checks by allowing flexibility in $\hat{\partial}$ and θ . In order to cover most possibilities, the transformation also ensures that $\hat{\partial}$ and θ are specified as ‘*’, **node()**. Once it completes the matching process, it confirms some q , which are then stored in the **semanticXPath** list (Lines 3:1-3:3).

Function 3 (shown below) then moves on to verify p_2 only if there is a valid p_1 . If p_1 is not valid, there is no reason to check p_2 . In addition to this, the transformation becomes invalid if either p_1 or p_2 is invalid. To ensure that p_2 is valid, it relies on the **semanticXPath** (Lines 3:4 – 3:7) which must not be empty. When checking for p_2 , it considers several checks for ϑ and θ specified as ‘*’ or **node()**. Based on the hierarchies that enclose ϑ , the unique paths associated with p_2 are identified accordingly. Once all valid elements, ϑ and θ have been matched, q is identified as p_2 and q is put into the **semanticXPath** list.

Function 3: List semanticPathComplement (List semanticXPath, String p)

Let β be fragment contained ' \dots ', $\partial \in \beta$ be query condition element, $\theta \in \beta$ be target element, θ be branching element, tempQ and tempList be empty list, p_1 be right branching path w.r.t $p_1 = p$ exclude (∂ and ' \dots '), Q be the list of unique path q, p_2 be left branching path w.r.t $p_2 = p$ exclude fragment from ' \dots ' to θ , $O = \{*, node()\}$

```

3:1 For each q in Q
3:2   push q to semanticXPath when ( $\partial \in o$  && valid elements including  $\theta$  in  $p_1$  match non-target elements in q
    &&  $\theta$  is not in q ) ||  $\partial$  &&  $\theta$  in  $p_1$  match elements in q &&  $\partial$  is target element in q)
3:3 End Loop
3:4 If semanticXPath is not NULL Then
3:5   For each q in Q
3:6     push q to semanticXPath when ( $\partial \in o$  && valid elements &&  $\theta$  in  $p_2$  match non-target elements in q &&  $\partial$ 
    is not target element in q ) || (valid elements &&  $\theta$  in  $p_2$  match elements in q &&  $\partial$  is target element in q)
3:7 End Loop
3:8 If  $\theta$  and  $\partial$  exist in separate q in semanticXPath Then
3:9   If (minimum(occurrence) of ( $\theta / \partial$ )  $\geq 1$  found in list C) Then
3:10    semanticXPath =  $p_1$ 
3:11   Else semanticXPath =  $p$ 
3:12 Else semanticXPath = NULL
3:13 Return semanticXPath

```

Once the semanticXPath has been built with unique paths that have been identified for p_1 and p_2 respectively, the transformation then eliminates ∂ from p . There are two identified unique paths q in the semanticXPath list. One represents the path of query condition and the other represents the path of selected information.

To remove the condition element, SPCom needs to check the occurrence constraint of a condition held by ∂ under parent θ using information in the list of constraints of elements C . The condition ∂ can be removed from p only if its minimal occurrence is greater than 0 (Lines 3:9 – 3:10). The condition ∂ cannot be removed from p only if its minimal occurrence is between 0 and 1 (Lines 3:11). The transformation detects a semantic conflict (Definition 4.6), and applies the semantic conflict detection rule in Section 4.2.4, when it detects semanticXPath with NULL (Line 3:12). It finally returns the semanticXPath list to the main algorithm (Line 3:13).

4.2.4 Semantics Conflict Detection

The satisfactory XPath query study for XPath query has been proposed by Groppe and Böttcher [2005], such that during the transformation the structure of XPath query can be detected with a conflict (unsatisfactory) that will produce an empty

result set. We show how this can be achieved by using Q and C . The difference between the work here and existing work [Groppe & Böttcher 2005], is that we incorporate the semantics conflict detection in our semantic transformation rule. The existing work does not provide a transformation of XPath query if no semantic conflict is detected.

Definition 4.6. (Semantic Conflict Detection - SCD) Semantic conflict detects conflicts of structure and element names specified in an XPath query by using unique path Q and constraints of elements in list C during the transformation process.

We develop a guideline for semantic conflict detection, namely SCD to assist with the termination of SPE, SPCon, and SPCom transformations. The algorithm of SCD is part of functions 1, 2 and 3.

Let us consider the following parameters:

- p is a given XPath query
- ε is ‘*’, ‘.’, node() or valid element in a location step of p
- Q is a list of unique paths derived from given schemas
- C is a list of constraints of elements

SCD Rule. The semantic conflict exists if one of the following rules is satisfied

1. elements in p do not appear in any q where $q \in Q$
2. ε exists in p so ε does not satisfy a path fragment in any q where $q \in Q$

The **SCD** semantic rule simply checks the correctness of the structure in p whereby if valid elements exist in p , they must match the same set of elements in a q first. Thereafter it matches ε based on the patterns that enclose ε in p to identify q . Here, for example, is an XPath query `dblp/*/*/tns`

This query will be matched to all $q \in Q$ based on the elements **dblp** and **tns** first. However, the matching process cannot locate **tns** in any $q \in Q$. As a result, a conflict of element non-existence has been detected in p .

In SCD, we do not develop an independent algorithm; instead, it is implemented within each semantic transformation as described earlier in semantic path transformations..

4.3 Summary

In this chapter, we have proposed: (1) a methodology to derive semantics provided in XML Schema and (2) semantic path transformation typologies including semantic path expansion, semantic path contraction and semantic path complement.

In the semantic derivation methodology, the semantics are classified and then divided into two lists including a unique path list and constraints of elements list. Ultimately, two sets of essential information are produced: a unique path and constraints of elements lists.

Once the essential information has been derived, we then propose the first semantic transformation category which is semantic path transformation; this consists of semantic path expansion, semantic path contraction and semantic path complement.

The semantic path expansion typology transforms a path fragment '//', '*' or 'node()' into a sequence of elements if and only if the XPath query can match a single unique path. The transformation takes into consideration any element labelled with '*', node(), or a valid element tag name. The transformation must first satisfy the proposed semantic rule. The rules are then translated into a function, namely **semanticPathExpansion** as a guideline.

The semantic path contraction typology transforms fragments that have elements labelled with '*' or node () in the user XPath query into fragment '//'. This can be true only when the XPath query is matched to multiple unique paths that have the same target element. This means that an element labelled with '*' represents multiple different path fragments within the multiple identified unique paths. The transformation must satisfy the proposed semantic rules, which are then translated to a function, namely **semanticPathContraction**.

The semantic path complement typology transforms path fragments with elements labelled with ‘..’ in the user XPath query into a sequence of elements in an identified unique path. When ‘..’ is specified in an XPath query, ‘..’ expects to associate with a condition element to filter information. The goal of this transformation is to eliminate the condition fragment including ‘..’ using unique paths in Q and also an occurrence constraint from the constraints of elements list C . The transformation must satisfy the proposed semantic rule followed by a function, namely **semanticPathContraction** to remove the condition fragment.

During the transformation, we have also integrated a technique to detect semantic conflicts that may return an empty query response. In this way, the transformation can immediately provide an answer without needing to complete the transformation process and avoid accessing the database unnecessarily. In the implementation, we demonstrate that the conflict detection makes a significant contribution to query transformation and ultimately boosts performance.

Chapter 5

Semantic Transformations for XPath Queries specified with XPath Axis

XPath queries are essentially composed of a succession of axes defining the navigation from a current context node. Among the XPath query axes family, *child*, *descendant*, *parent* and *self* can be optionally specified using the path notations $\{/,//,...\}$ which have been commonly used. Axes such as *following*, *preceding*, *ancestor*, *ancestor-or-self*, *following-sibling* and *preceding-sibling* have unique functionalities which provide different required information that cannot be achieved by others. However, XPath query optimization using schema constraints do not yet consider the XPath axes family.

The performance of queries denoting the same result by means of different axes may significantly differ. The difference in performance can be affected by some axes, but this can be avoided. The aim of these proposed semantic transformations is to modify the structures of XPath queries by eliminating the XPath axes to improve query performance.

For a complete solution, this chapter proposes several semantic transformations to transform XPath queries specified with XPath axes *{following-sibling, preceding-sibling, following, preceding, ancestor, ancestor-or-self, parent, descendant, descendant-or-self, self, child}* [W3C, 1999; 2007a; 2010] for optimization purposes.

Unique paths in list Q and constraints of elements in list C , which have been derived from XML Schemas [W3C, 2004a; 2004b] in Section 4.1 Chapter 4, are used to propose the transformations. Recall that list Q contains a list of unique paths q where each q is a sequence of elements that express a path from the root to a particular element in a given XML Schema. Only the parent-child “/” relationship among the elements is in q . List C contains a list of c where each c contains the element and its constraints specified in the XML schema.

5.1 Query Transformation Direction and Defining Parameters

This section achieves the following tasks:

1. The tree pattern (refer to Definition 3.11) represents the structure of XPath queries. The directional approach used by semantic transformations for XPath queries may start from the left-most location step (refer to Definition 3.6) or the right-most location step [Bashir and Boulos, 2005; Furfaro & Masciari, 2003]. Such a directional approach is adopted in this research, which is sometimes referred to as a right-most or left-most direction. The first task of this section is to describe the directional approach of semantic transformations.
2. The proposed semantic transformations in this chapter share a number of parameters; these parameters are referred to as global parameters. The second task of this section is to define a set of global parameters.

5.1.1 Transformation Direction for XPath Query

As mentioned in the previous section, the transformation of XPath queries uses information such as unique paths q in Q and constraints c in C . This section proposes that the query transformation direction start from the right-most XPath query; that is the target element, which is the right-most element and is located in the right-most location step in an XPath query. It is easy to match the target element in an XPath query to the target element in q to first identify all possible q . The remaining elements in XPath can then easily be matched to those in the identified q . Such a transformation direction is explained in more detail with examples in this section.

An XPath query may be specified with an axis to navigate to a specific location, for example, consider the following XPath query

$$p = i/k/\text{following-sibling}::*$$

In this XPath query, location steps i and k are specified with no axis names and the right-most location step ***following-sibling::**** is specified with axis name ***following-sibling***. This research proposes a translation of an XPath axis for all the elements in the XPath query that are specified without an axis before the transformation as shown below.

$$i/k/\text{following-sibling}::* \text{ translates to } \text{child}::i/\text{child}::k/\text{following-sibling}::*$$

The transformation goal is to obtain the semantic XPath query/queries. It can be a single semantic XPath query or multiple XPath queries, which is referred to as a *semanticXPath* list.

Figure 5.1 shows how the *semanticXPath* list is first produced using the right-most direction.

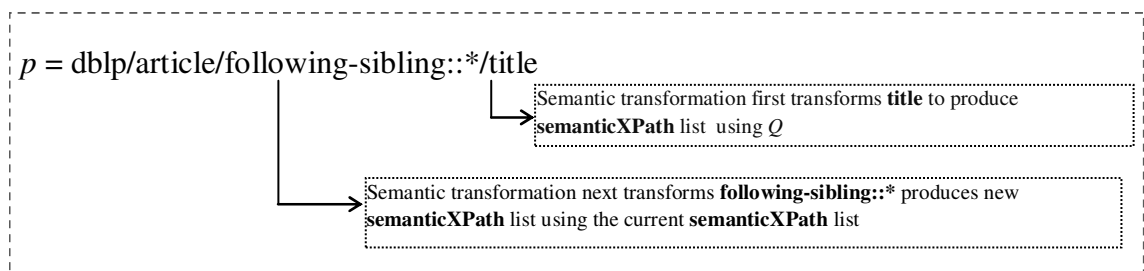


Figure 5.1 Sample of Semantic Transformation with a Bottom-up Approach

As the **title** is in the right-most location step, which is also known as the target location step (refer to Definition 3.9), the *semanticXPath* list is always empty and information in the *Q* list will be used to produce the *semanticXPath* list. The next location step is *following-sibling::**, which is a non-target location step (refer to Definition 3.10). Therefore, *following-sibling::** is transformed using the *semanticXPath* list instead of a unique path list, as the *semanticXPath* list is not empty. For example, consider the transformation of *p* in Figure 5.1. The transformation starts from the **title** using information in *Q* list and the result would be stored in *semanticXPath* list. It then moves on to transform the next location step, which is *following-sibling::** in which it will use the *semanticXPath* list because at this point, the *semanticXPath* list is not empty.

The semantic transformation relies on input information including unique paths in list *Q* or *semanticXPath* (the arrival of *semanticXPath* has been explained above), and constraints and their values of elements in list *C*. In addition to this provided information, the transformation also explores the context position functions such as *position()*, *last()* and context position value [Brantner, 2005; W3C, 1999; 2010].

5.1.2 Defining Global Parameters

The semantic transformations proposed in this chapter utilize the same terms and information related to XPath query, lists of *Q* and *C* to progress to semantic rules. This section defines a set of global parameters that reference to the terms and information which are used to define the semantic transformation rules and translate them to the algorithms throughout this chapter.

- β is a transforming location step in an XPath query *p*.
- α is an axis.
- ϵ is a node-test where $\beta = \alpha::\epsilon$ and ϵ is ‘*’, *node()* or a valid element tag
- ∂ is a context element that occurs in the location step next to β , which must appear on the left of β .
- ϑ is the parent element of ∂ that occurs in the location step next to ∂ which must appear on the left of ∂ .

- ∞ is an infinite occurrences of an element. It is used to denote a maximal occurrence of an element.
- Q is list of unique paths derived from schemas.
- C is the list of elements, their constraint names, and values of the constraints.
- S (abbreviation for *semanticXPath* list) is a list that contains semantic XPath query/queries.

5.2 Semantic Transformation for Following- or Preceding-sibling Axis

The semantic transformation aims to remove a *following-* or *preceding-sibling* axis using information in lists Q and C . Depending on the location of the *following-* or *preceding-sibling* axis in an XPath query, S is used when S is not empty.

Firstly, we propose semantic rules to transform a given XPath query to its equivalent semantic XPath query. Secondly, we translate the proposed rules to a semantic transformation algorithm.

It is possible to produce a single semantic XPath query for a given XPath query specified with *following-* or *preceding-sibling* axis. However, the majority of transformations expect multiple XPath queries.

Before the semantic rules are proposed below, the XPath query `*/article/author/following-sibling::*` is used to shade light on how semantics are used to obtain the semantic XPath queries by eliminating the axis. This given XPath query finds all the followed siblings of the first occurrence of the article **author**. By using information in Q , the unique path `dblp/article/author` selects the article **author** and its following unique paths that select the siblings of the article **author**, e.g. `dblp/article/title`, `dblp/article/chapter`, etc., are retrieved as semantic XPath queries. Depending on the occurrence constraint, located in C , of article **author**, the unique path that selects the **author** may be excluded from the semantic XPath queries. Otherwise the context index position function such as `position()` is used by the unique path that selects article **author** to reassure the last occurrence of **author** is

not selected. This example will be revisited later on in great details by the proposed semantic transformation rules.

Prior to the proposal of semantic rules, it should be kept in mind that if the transforming location step is the right most location step, also known as target location step, in the XPath query then Q will be used to derive $S_{(\text{temp})}$. If the transforming location step is not the right most location step in the XPath query, also known as a non-target location step, S is not empty, and hence S is used to derive $S_{(\text{temp})}$. $S_{(\text{temp})}$ is a temporary list that contains semantic XPath queries. Finally $S_{(\text{temp})}$ is used to derived S where the transformation rule will decide if the last or first occurrence of context element is selected. This important note is also applied to semantic rules proposed in section 5.3.

For each of the *following-* or *preceding-sibling* location steps (referred to as the transforming/transformation location step) in an XPath query p , the semantic transformation rule called **ST_{fps}** is now proposed.

ST_{fps} Rule. Semantic XPath query S is derived as follows:

- a. When ϵ and ∂ are '*' or node() and ϑ is a valid element name, then locate q_k in Q or S , where q_k must contain ϑ as a target element. The child element of ϑ is the first child element or the last child element of ϑ .
- b. When ϵ and ϑ are '*' or node() and ∂ is valid element name, then locate q_k in Q or S where q_k must contain ∂ as a target element.
- c. When ∂ and ϑ are '*' or node() and ϵ is valid element name, then locate q_k in Q or S where q_k must contain ϵ as a target element.
- d. When ϵ is '*' and ∂ and ϑ are valid element names, then locate q_k in Q or S where q_k must contain ∂ as a target element.

In summary, the $S_{(\text{temp})}$ is derived with given q_k as follows

$$S_{(\text{temp})} = \{q_k, \dots, q_m\} \begin{cases} \text{If } \alpha \text{ is following-sibling then } k < m. \\ \text{If } \alpha \text{ is preceding-sibling then } k > m \end{cases}$$

By now, the $S_{(\text{temp})}$ contains a set of semantic XPath queries that has been transformed by the semantic rule as proposed above. However, the first semantic XPath in $S_{(\text{temp})}$ may select the first or last context element, which needs to be further transformed so that the first or last occurrence is excluded from the final result. The first occurrence of the context element is not selected when the *following-sibling* axis is used. The last occurrence of the context element is not selected when the *preceding-sibling* axis is used. To achieve this, the context function ($\text{position()} > 1$ or $\text{position()} < \text{last}()$) is used as below so that the final result S is derived:

Derivation of S using $S_{(\text{temp})}$

$S = S_{(\text{temp})}$ where q_k will not select the first occurrence of ∂ by placing index context function $[\text{position()} > 1]$ or $[\text{position()} < \text{last}()]$ (for *following-* or *preceding-sibling* axis) on ∂ when the occurrence set for ∂ is between 1 and ∞ . If occurrence of ∂ is set between 1 and 1 then q_k is removed from S as q_k produces only one occurrence of ∂ which is not required for the collection set.

List S contains information that occurs on the right or left (depending on α is following or preceding axis) of the first occurrence of ∂ selected by q_k .

The following examples show how S is ultimately derived.

$$p = */\text{child}::i/\text{preceding-sibling}::* \quad Q = \{q_1, \dots, q_{12}, \dots, q_{17}, \dots, q_{19}, q_{20}, \dots, q_{22}\}$$

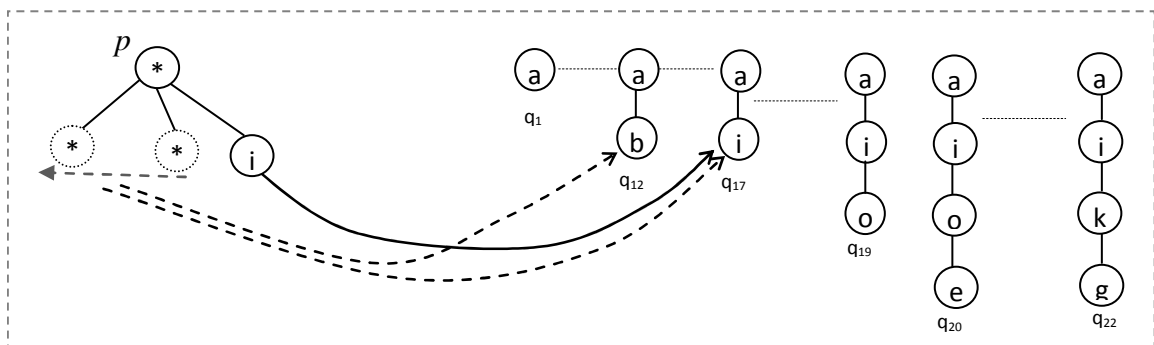


Figure 5.2 Semantic Transformation for Preceding-sibling Axis Query

An XPath query tree p and a set of unique paths $Q = \{q_1, \dots, q_{22}\}$ are given as shown in Figure 5.2. The left directional arrow indicates that the selected elements must

occur on the left of the last occurrence of i in the XML data tree. The dashed arrow lines (from p to q) indicate the information to be selected that must occur on the left of element i in p , which can be accomplished by unique paths q_{12} to q_{17} . The thick lines (from p to q) indicate the context element in the first identified q . Note that both dashed and thick lines map to the same element i in q_{17} ; this is to indicate that element i may have many occurrences. If this is the case, only the last occurrence of i is not selected.

By following rule (a) in $\mathbf{ST}_{(\text{fps})}$, q_{17} has i as a target element. q_{12} to q_{17} share the same parent a ; hence, $S_{(\text{temp})} = \{q_{12}, \dots, q_{17}\}$. The occurrence constraint of element i needs to be verified against list C to decide if $[\text{position()} < \text{last}]$ is added to i in q_{17} or q_{17} should be removed from $S_{(\text{temp})}$ to produce S .

Assuming that list C indicates that the occurrence of a/i is between 1 and infinity (denoted as ∞) then,

$$S = S_{(\text{temp})} \text{ and } q_{17} \text{ is now } q_{17} = a/i[\text{position()} < \text{last()}]$$

Let us consider an example based on the DBLP schema (Appendix 1) that is used to derive lists Q and C .

Example. A requirement selects all the siblings of article and author that follow it in the DBLP database.

XPath query $p = */\text{article}/\text{author}/\text{following-sibling}::*$

Based on given p , the values of the required parameters are now extracted such that

- β is *following-sibling::**
- ∂ is **author**
- ϵ is *****
- ϑ is **article**
- Q is $\{\text{dblp}, \text{dblp}/\text{article}/@\text{key}, \text{dblp}/\text{article}/@\text{mdate}, \text{dblp}/\text{article}/\text{author}, \text{dblp}/\text{article}/\text{title}, \text{dblp}/\text{article}/\text{year}, \dots, \text{dblp}/\text{article}/\text{url}, \dots\}$
- C is $\{\text{dblp sequence}, \dots, \text{article}/\text{author occurrence } 1 \infty, \dots\}$

- S is empty

By following rule (d) in $\mathbf{ST}_{(\text{fps})}$ rule, p has *following-sibling::** as the target location step; therefore, S is empty and Q will be used to produce $S_{(\text{temp})}$ where $S_{(\text{temp})} = \{\text{dblp/article/author, dblp/article/title, .., dblp/article/url}\}$ because **author** has **title**, **year**, ..., **url** as siblings followed (or were to the right of) **author**.

Next, the transformation also finds that C contains a list item '**article/author occurrence 1 ∞** '. This means that each **article** must have at least one **author** as a minimal occurrence and an infinite number of authors as the maximal occurrence. Therefore, there is a need to add $[\text{position()}>1]$ to **author** in the unique path that selects the author information. This context function will select all the authors of each article except the first author. The transformation produces the result as follows:

$$S = \{\text{dblp/article/author}[\text{position()}>1], \text{dblp/article/title}, \dots, \text{dblp/article/url}\}$$

Function 1. transformFollowPrecedingSibling translates the semantic transformation rules for transforming XPath query specified with *following-* or *preceding-sibling* axis.

The function is called from the main algorithm when *following-* or *preceding-sibling* location step is detected in a given XPath query.

Let us remind readers that any parameters that are not defined in the algorithms from this point onward should be the global parameters which have been defined earlier in the Chapter.

The function determines whether the transforming location step is a target or a non-target location step. This is done by checking the *SemanticXPath*. If it is empty (Line 1:1), list Q will be used for transformation otherwise *SemanticXPath* is used for transformation.

When *followingSibling* (Line 1:2) is not NULL, which means the transforming location step is specified with *following-sibling* axis, the function sets the starting point and direction for searching unique paths as well as the label of the axis.

Otherwise, the function checks for *preceding-sibling* axis, starting point, search direction and axis label (Line 1:3).

Based on the identified axis and search direction, the transformation determines whether the transforming location step is a target location step or a non-target location step. The transformation first builds up the *tempList*.

Function 1: List transformFollowPrecedingSibling(String eContext, String followingSibling, String precedingSibling, String parentContext, List semanticXPath)

```

    Let tempQ and tempList be empty lists, O be occurrence of eContext,  $\Upsilon$  be an initial point of traversing and searching
    Direction,  $\omega = \{*, node()\}$ , Q be the list of unique path q, C be the list of constraints of elements
1:1  IF semanticXPath is empty THEN tempQ = Q ELSE tempQ = semanticXPath
1:2  IF followingSibling is not NULL THEN  $\Upsilon$  starts from top of list in tempQ Set axis = following-sibling
1:3  ELSE precedingSibling THEN  $\Upsilon$  starts from the end of list in tempQ and Set axis = preceding-sibling
1:4  IF semanticXPath is empty THEN
1:5    For each q, using  $\Upsilon$ , in tempQ
1:6    IF ((parentContext not in  $\omega$  && found in q) && target element in q is eContext ) THEN
1:7      For each q in Q
1:8        IF ((q has parentContext && (target element as (sibling of eContext || eContext) in q)
           || ((q has parentContext && eContext is '*' target element as in q is axis) THEN push q to tempList
1:9        ELSE IF ((parentContext is in  $\omega$  && target element in q is eContext ) THEN
1:10       For each q in Q
1:11       IF ((target element (as sibling of eContext || as eContext) in q) && axis element is in  $\omega$  ||
           (( target element in q is axis element) THEN push q to tempList
1:12  ELSE
1:13  FOR each unique path q, using  $\Upsilon$ , in tempQ
1:14    IF (axis element is not in  $\omega$  and found eContext in q) THEN
1:15      FOR each current q in tempQ IF q contain axis element THEN Push q to tempList
1:16    ELSE push q to tempList
1:17  IF followingSibling is not NULL THEN
1:18    IF O of eContext is (1,  $\infty$ ) THEN add [position()>1] to eContext of first q in tempList
1:19    ELSE IF O of eContext is (1,1) THEN remove first q from tempList
1:20  ELSE IF precedingSibling is not NULL THEN use tempList
1:21    IF O of eContext is (1,  $\infty$ ) THEN add [position()<last()] to eContext of last q
1:22    ELSE IF O of eContext is (1,1) THEN remove last q from tempList
1:23  ELSE IF O of eContext is (0,1) semanticXPath = 'Retain'
1:24  IF (tempList is empty) semanticXPath = NULL
1:25  ELSE IF semanticXPath Not 'Retain' THEN semanticXPath= tempList;
1:26  RETURN semanticXPath

```

When semantic transformation detects a transformation location step as a target location step, it locates all the unique paths by considering the following scenarios: first, parent of context element is not '*' or node() and context element has a valid

label name or '*' or node(). Second parent of context element is '*' and context element has a valid label name or '*' specified with indicated axis (Lines 1:4-1:12).

When semantic transformation detects a transformation location step as a non-target location step, it locates all the unique paths by considering the axis and context elements as a combination (Lines 1:13-1:16).

Once the transformation produces a *tempList*, it then decides to add the context position function *position()* or *last()* when minimal and maximal values of *occurrence* constraint of the context element are 1 and ∞ (Lines 1:18, Line 1:21). The unique path *q* is removed from the potential list of semantic XPath queries when the minimal and maximal values of *occurrence* constraint of context element are 1 (Line 1:19, Line 1:22).

The algorithm also handles the case of minimal and maximal values of occurrence constraint of 0 and 1. It is not possible to detect that the first context element occurs in the database unless it accesses the XML document data during the transformation. In the case of an *occurrence* constraint that has minimal and maximal occurrences of 0 and 1, the *semanticXPath* is set to 'Retain' acknowledges to the main algorithm that the location step is valid but retains its original specification (Line 1:23). This means that the XPath query has not been transformed.

The semantic transformation is also able to detect conflicts (Line 1:25), which results in an empty data set by detecting the *tempList*. This will boost performance significantly as the query does not need to access the database before it returns the answer to the user.

If the *semanticXPath* has no information and the *tempList* is not empty, then the *semanticXPath* is assigned with the *tempList* and it is returned to the main algorithm (Line 1:26).

5.3 Semantic Transformation for Following or Preceding Axis

The *following* or *preceding* axis navigates the information on the right or left of the context element as well as the information following or preceding where the context element begins or ends.

To obtain the semantic equivalent XPath query, the semantic transformation is now proposed for the *following* or *preceding* axis. The semantic transformation aims to remove a *following* or *preceding* axis by using list Q of unique paths and list C of constraints of elements. The semantic transformation for these two axes, in most cases, produces more than one semantic XPath query (referred to as a set of semantic XPath queries).

The XPath query ‘dblp/article/title/preceding::*’ is now used to shade light on how semantics are used to obtain the semantic XPath queries by eliminating the axis preceding. This given XPath query finds everything that occurs in front of the first occurrence of article **title**. The unique path ‘dblp/article/title’ and all unique paths in Q that select information occurs in front of the first occurrence of article **title**. Basically it is looking at all the unique paths that occur in front of the unique path ‘dblp/article/title’ and all unique paths that select the siblings of article **title**. Furthermore, depending on the occurrence constraint, located in C , of article **title**, the transformation also needs to use the context index position function *position()* so that it re-assures the last occurrence of **title** and everything following it will not be selected. This example will be revisited later on in great details by the proposed semantic transformation rules.

For each of the location steps specified with *following* or *preceding*, also known as transforming location steps, in an XPath query p , the semantic transformation rule called **Steps** is now proposed.

Steps Rule. Semantic XPath query S is derived as follows:

- a. When ϵ is ‘*’ or *node()* and ϑ and ∂ are valid element names, then locate q_k in Q or S where q_k contains ∂ as a target element.

- b. When ε and ϑ are '*' or node() and ∂ is a valid element name, then use ∂ to locate q_k in Q or S where q_k contains ∂ as a target element.
- c. When ε and ∂ are '*' or node() and ϑ is a valid element name, then use ϑ to locate q_k in Q or S where q_k has ϑ which also has a child as a target element and is the first or last child of ϑ .
- d. When ε is a valid element name and ϑ and ∂ are '*' or node(), then locate q_k in Q or S where q_k has ε as a target element.

$$S_{(\text{temp})} = \{q_i, \dots, q_k, \dots, q_z\}$$

If α is the **preceding** axis then q_i, q_{k-1} are all unique paths that are in front of q_k . q_k, \dots, q_z must contain ϑ and their target elements are children of ϑ and its siblings occur in all q leading q_k . $i = 0$ (at the top of Q), $i < k < z$.

If α is the **following** axis, then q_i, \dots, q_{k+1} are all unique paths that are located after q_k . q_k, \dots, q_z and must contain ϑ ; their target elements are children of ϑ and its siblings occur in all q following q_k . $i = \text{length of } Q$, $i > k$ and $z > k$

Derivation of S using $S_{(\text{temp})}$

When the occurrence of ∂ in q_k is between 1 and ∞ , context function [position() >1] or [position() < last()] is replaced on ∂ and [position() = 1] or [position() = last()] on ϑ in q_k . $q_{z+1} = q_k$ and [position() >1] or [position() < last()] is placed on ϑ in q_{k+1}, \dots, q_{z+1} .

When the occurrence of ∂ is between 1 and 1, context function [position() >1] or [position() < last()], depending on α , is placed on ϑ in q_k, \dots, q_z or q_k, \dots, q_z .

These functions are important as they handle the role of not selecting the first or last occurrence of ∂ and the siblings occur before or after the first ∂ in the whole document.

$S = S_{(\text{temp})}$ where q_k, \dots, q_z are modified as described above.

The following example show how S is ultimately derived. An XPath query $p = */child::i/child::k/preceding::*$ and a set of unique paths $Q = \{q_1, q_1, q_3, \dots, q_4, q_5, q_6, q_7, \dots, q_{22}\}$ is given as shown in Figure 5.3.

The left-most tree represents the XPath query p . The left-pointed and vertical-pointed arrows show the direction of selected information in the XML data tree that is expected to occur before the last context element k . The mapping arrows from p to Q indicate the information to be selected. By following rule **a** in **Steps**, both the parent and context elements are valid for deriving $S_{(temp)}$. As a result, $S_{(temp)} = \{q_6, \dots, q_2, q_1\}$ has been derived where q_5 is the unique path that selects the context element k .

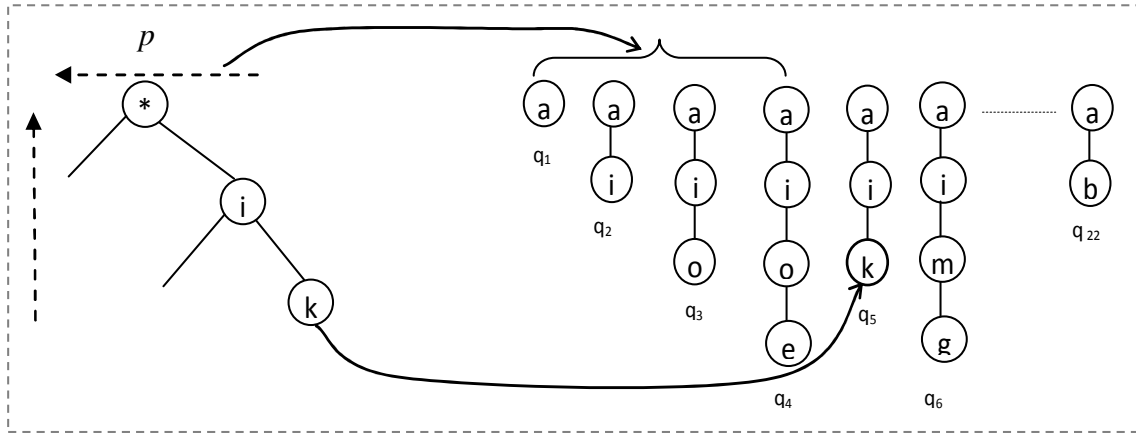


Figure 5.3 Semantic Transformation for Preceding Axis Query

S now can be derived based on the presence of the **preceding** axis in p by checking the occurrence of i/k in list C . Suppose that the occurrence of i/k is between 1 and ∞ , then

$S = S_{(temp)}$ where q_5 and q_6 are replaced as follows:

$$q_5 = a/i[\text{position()} = \text{last()}]/k[\text{position()} < \text{last()}], a/i[\text{position()} < \text{last()}]/k$$

$$q_6 = a/i[\text{position()} < \text{last()}]/m/g$$

Let us consider an example based on the DBLP schema that is used to derive lists Q and C .

Example. A requirement selects all the members preceding the article and title in the DBLP database.

A user defined XPath query $p = \text{dblp/article/title/preceding::}^*$

Based on given p , the values of the required parameters are extracted as follows:

- β is ***preceding::****
- ∂ is **title**
- ϵ is *
- ϑ is **article**
- Q is {dblp, dblp/article/@key, dblp/article/@mdate, dblp/article/author, dblp/article/title, dblp/article/years,..., dblp/article/url, dblp/inproceedings,...,}
- C is {dblp sequence, dblp/article sequence, dblp/article occurrence 1 ∞ ...,}
- S is empty as the preceding location step is specified as a target location step in p

By following rule **a** in **Steps Rule**, $S_{(\text{temp})}$ is derived as follows:

$$S_{(\text{temp})} = \{\text{dblp, dblp/article/@key, dblp/article/@mdate, dblp/article/author, dblp/article/title, dblp/article/years,..., dblp/article/url}\}$$

The occurrence of **article/title** is between 1 and 1 and is located in list C . The context function $[\text{position()} < \text{last()}]$ is added to the **article** in the unique path **dblp/article/title** and all the unique paths that appear behind it in $S_{(\text{temp})}$. The result of S is as follows:

$$S = \{\text{dblp, dblp/article/@key, dblp/article/@mdate, dblp/article/author, dblp/article}[\text{position()} < \text{last()}]/\text{title, dblp/article}[\text{position()} < \text{last()}]/\text{years,..., dblp/article}[\text{position()} < \text{last()}]/\text{url}\}$$

Function **2 transformFollowPreceding** translates the semantic transformation rules. Function 2 is called when the transforming location step in an XPath query is specified with *following* or *preceding* axis.

The transformation first determines if the Q or *semanticXPath* list is used. This is when the *following* or *preceding* location step in the XPath query is the target or non-target location step (Line 2:1). The *tempQ* list is set.

The transforming location step then detects whether the *following* or *preceding* axis is present so that the initial point and traversing direction for retrieving unique paths are set (Lines 2:2 – 2:3).

Function 2: List transformFollowPreceding(String eContext, String following, String preceding, String parentContext, List semanticXPath)

```

Let tempQ and tempList be empty lists, O be occurrence of eContext, Q be the list of unique paths q,
Y be an initial point of traversing and searching direction,  $\omega = \{*, node()\}$ , C be the list of constraints of elements
2:1 IF semanticXPath is empty THEN tempQ = Q ELSE tempQ = SemanticXPath
2:2 IF following is not NULL THEN Y starts from top of list in tempQ Set axis = following
2:3 ELSE it is precedingSibling THEN Y starts from the end list in tempQ and Set axis = preceding
2:4 IF semanticXPath is empty
2:5   FOR each q, using Y, in tempQ
2:6     IF ((q contains parentContext && eContext is the target element in q) || axis element is the target element in q) THEN
2:7       FOR each q in tempQ
2:8         IF (axis element is in  $\omega$  || (eContext found in q && axis element is target node in q) || (eContext is in  $\omega$  &&
2:9           axis element is target node in q)) THEN push q to tempList
2:10      END LOOP
2:11    END LOOP
2:12 ELSE
2:13 FOR each unique path q, from top of list, in semanticXPath
2:14   IF ((eContext is in  $\omega$  && a target element in q) || (eContext is in  $\omega$  && following found in q)) THEN Push q to tempList
2:15 END LOOP
2:16 IF (tempList is empty) THEN semanticXPath = NULL; Exist
2:17 IF following && tempList != NULL THEN q = first q that has both eContext and parent from tempList
2:18 ELSE IF preceding && tempList != NULL THEN q = last q that has both eContext and parent from tempList
2:19   Get the minOccurrence and maxOccurrence values of eContext node
2:20 IF ((following && minOccurrence >= 1 && maxOccurrence > 1) THEN
2:21   Append position()>1 to eContext location step and position()= 1 to parentContext in current q
2:22   Duplicate current q and append position()>1 to parentContext of duplicated q and add it to tempList
2:23   Append position()>1 to parentContext in all q that comes after current q
2:24 ELSE IF (preceding && minOccurrence >= 1 && maxOccurrence > 1) THEN
2:25   Append 'position()<last()' to eContext and 'position()=last()' to parentContext in current q
2:26   Duplicate current q and append 'position()<last()' to parentContext of duplicated q and add it to tempList
2:27   Append position()<last parentContext in all q that comes before current q
2:28 ELSE IF ((following && minOccurrence=1 && maxOccurrence = 1) ||
2:29   (preceding && minOccurrence=1 && maxOccurrence = 1)) THEN
2:30   Append (position()>1 or position()<last()) to parentContext in current q
2:31   and all q that have target elements are siblings of eContext
2:32 ELSE IF ((minOccurrence=0) || (minOccurrence=0) ) THEN tempList = 'Retain'
2:33 semanticXPath = tempList;
2:34 RETURN semanticXPath

```

The first part in the transformation is to locate the unique paths q for *following* axis specified in p (Lines 2:4-2:8). The first matched unique path q in Q is located where q contains a valid parent element and valid context as the target element or the context element is '*' or node() and the parent element is a valid element name. Otherwise, it checks for a valid context element name and parent element is '*' or node(). The transformation continues to retrieve all the unique paths q that produce information following or preceding information produced by q . The *tempList* is built.

When the context element is not a target element, the transformation locates the q from the existing *semanticXPath* list in which q has only the parent element and the context is a non-target (Lines 2:9-2:12). The *tempList* is built here.

When the *tempList* is empty, this indicates that a conflict has been detected and no further transformation is needed. The *semanticXPath* returns to the main program as NULL that allows the main program to handle the configured message set in the main program (Line 2:13).

If the *tempList* is not empty, the transformation will continue to produce the *semanticXPath* list. The occurrence constraint of the context element is retrieved from list C (Line 2:16). The *tempList* list produced for the *following* axis is to use the context functions *position()* that does not select the first occurrence of the context element if the minimal and maximal values of *occurrence* constraint of context element are 1 and ∞ (Lines 2:17 – 2:20). The occurrence constraint values are also applied to produce the *tempList* list for the *preceding* axis, and the index context functions such as *position()*, and *last()* are used for the *preceding* axis (Lines 2:21-2:24).

When the minimal and maximal values of the *occurrence* constraint of the context element are 1, the context functions *position()* and *last()* ensure that the context element under the first or last parent is not selected (Lines 2:25-2:26).

When the minimal value of the *occurrence* constraint of the context element is 0, a '**Retain**' message is set in the *tempList*, which is subsequently assigned to the *semanticXPath*. This message tells the main program that the *following* or *preceding* location step in the XPath query is valid but has no new transformation (Line 2:27).

Finally, the valid semantic XPath queries are returned to the main algorithm (Line 2:29).

5.4 Semantic Transformation for Ancestor or Ancestor-or-self Axis

While *Ancestor* axis in an XPath query navigates information of all ancestors (parent, grandparent, etc...) of the context element, *ancestor-or-self* navigates information of all ancestors (parent, grandparent, etc..) of the context element and the context element itself [W3C, 2010]. An *ancestor* or *ancestor-or-self* axis may be specified with “*” or node().

Before the semantic transformation rule is proposal, the XPath query ‘//article/ancestor::dblp’ is now used to give a brief overview of how semantics are used in semantic transformation by eliminating XPath axis. This XPath query performs a task to select ancestor of **article** that must be **dblp**. By using list Q , the unique path that selects **article** is selected first. Because of **dblp** is the ancestor of **article**, any unique path that precedes the unique path of selecting **article** and also selects **dblp** is selected. This example will be revisited later on in great details by the proposed semantic transformation rules.

A set of semantic transformation rules called ST_{as} is now proposed for transforming an XPath query specified with an *ancestor* or *ancestor-or-self* axis.

Along with the proposal of semantic transformation rule ST_{dos} , an important note confirms that if the transforming location step is the right most location step, also known as target location step, in the XPath query then Q will be used to derive S . If the transforming location step is not the right most location step in the XPath query, also known as a non-target location step, S is not empty, and hence S is used to derive the new S . This important note is also applicable to all semantic rules in sections 5.5, 5.6, 5.7, 5.8 and 5.9.

ST_{as} Rule. Semantic XPath query S is derived as follows:

- When ε is '*' or node() and ϑ and ∂ are valid element names, then locate q_k in Q or S where q_k must contain ϑ and ∂ where ∂ is a target element.
- When ε and ∂ are '*' or node() and ϑ is a valid element name, then locate q_k in Q or S where q_k must contain ϑ that must have a child.
- When ε and ϑ are '*' or node() and ∂ is a valid element name, then locate q_k in Q or S where q_k must contain ∂ as the target element.

$$S = \{q_i, \dots, q_k\} \text{ where } i = 0 \text{ and } i \leq k$$

- When ε , ϑ and ∂ are valid element names, then locate q_k where ε must be a valid ancestor of ∂ .

$$S = \{q_k\} \text{ where } 0 \leq k$$

The following examples show how S is derived.

$$p = */child::i[child::o/ancestor::*] \quad Q = \{q_1, \dots, q_{12}, \dots, q_{17}, \dots, q_{19}, q_{20}, \dots, q_{22}\}$$

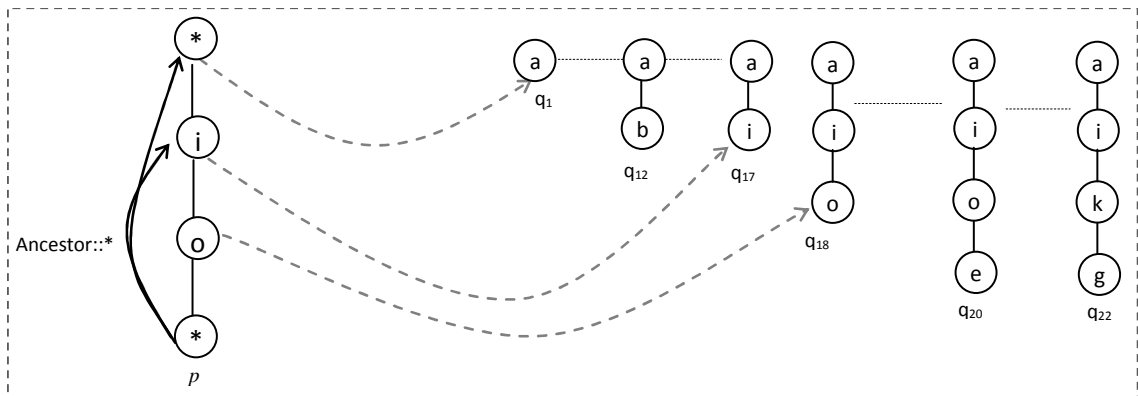


Figure 5.4 Semantic Transformation for Ancestor or Ancestor-or-self

An XPath query tree p and a set of unique paths $Q = \{q_1, \dots, q_{22}\}$ are given as shown in Figure 5.4.

Based on ST_{as} rule set, rule **a** is suitable because according to the given p , the value of ε is '*', the value of ϑ is o and the value of ∂ is i . For each element in p , except the

context element \mathbf{o} , the latter of which is mapped to the target element in any q in Q . In this case, q_{18} is found. Unique path q_{17} satisfies the presence of element \mathbf{i} , which is the target element and the parent of \mathbf{o} in q_{18} . Using q_{17} , the parent of element \mathbf{i} is \mathbf{a} . Any unique path that has an element label with \mathbf{a} as the target element and occurs before q_{17} is selected. q_1 is identified. q_{18} is not needed as it produces the information for the context element.

In this query, the result is a collection of information about the ancestors only.

$$S = \{q_1, q_{17}\}$$

Let us consider an example using the DBLP Schema that is used to derive lists Q and C .

Example. A requirement selects ancestry information of article titles as well as their own information in the DBLP database.

XPath query $p = */article/title/ancestor-or-self::*$

Based on given p , the values of the required parameters are extracted as follows:

- β is ***ancestor-or-self::****
- ∂ is **title**
- ϵ is *
- ϑ is **article**
- $Q = \{\text{dblp}, \text{dblp/article/@key}, \text{dblp/article/@mdate}, \text{dblp/article/author}, \text{dblp/article/title}, \dots, \text{dblp/article/url}, \dots, \text{dblp/www/url}\}$
- S is empty as ancestor-or-self location step is the target location step.

By using Q , a unique path that selects context information **title** for **article** is **dblp/article/title**, the next unique path that selects the **article** are **dblp/article/@key**, **dblp/article/mdate** and **dblp**.

$$S = \{\text{dblp}, \text{dblp/article/@mdate}, \text{dblp/article/@key}, \text{dblp/article/title}\}$$

Function **3. transformAncestors** translate the semantic transformation rules for transforming XPath query specified with *ancestor* or *ancestor-or-self* axis.

A *tempList* is built when an ancestor element is a non-target element and the context element is a valid element name that appears to be the target element, or *q* has an ancestor element as a non-target element, and the target element can be any element (Lines 3:1–3:5).

When the transforming location step is a non-target location step, the *tempList* is derived using *S*. The *tempList* is built by *q* that has an ancestor as a non-target element, and the context element which appears to be target element or *q* has an ancestor element as a non-target element, and the target element can be any child (Lines 3:6–3:10).

In the case where the *tempList* is NULL, a conflict is detected for the transforming location step (Line 3:11).

Function 3: List transformAncestors(String eContext, String eAncestor, List semanticXPath, Boolean ancestorOnly)

```

    Let tempList be an empty list,  $\omega = \{*, \text{node}()\}$ , Q be the list of unique paths q
3:1  IF semanticXPath is empty Then
3:2    FOR each unique path q in Q
3:3      IF ((eContext found as target element in q && (eAncestor found in q || eAncestor
          element is in  $\omega$  )) || (eContext element is in  $\omega$  && (eAncestor element found as target element in q) ) Then
3:4        push q to tempList
3:5    END LOOP
3:6  ELSE IF semanticXPath is not empty Then
3:7    FOR each q in semanticXPath
3:8      IF ((eContext element found as target element in q &&
          (eAncestor elemnt found in q || eAncestor element is in  $\omega$  ) ) Then
3:9        push q to tempList
3:10   END LOOP
3:11  IF (tempList is empty) semanticXPath = NULL
3:12  ELSE IF (tempList not empty && ancestorOnly is true)
3:13    construct a new list of semanticXPath q in tempList
3:14    Each new q produces information of each ancestor starting from the left most element in q to the eAncestor
3:15  ELSE IF (tempList not empty && ancestorOnly is false)
3:16    construct a new list of semanticXPath using q in tempList
3:17    Each new q produces information of each ancestor starting from the left most element in q to the eContext
3:18  RETURN semanticXPath

```

The *tempList* expects a unique path that has the context element as a target element. When the axis is ***ancestor***, unique paths are used to produce ancestry information which must be located as stated (Lines 3:12-3:14). When the axis is ***ancestor-or-self***, unique paths are used to produce ancestor-or-self information which must be located as stated (Lines 3:15-3:17). The final *semanticXPath* that produces information based on the ***ancestor*** or ***ancestor-or-self*** location step is returned to the main program (Line 3:18).

5.5 Semantic Transformation for Parent Axis

The ***Parent*** axis in an XPath query navigates information of the element that immediately occurs before the target element [Ozcan et al, 2008; W3C, 2007a; 2010; Yuen & Poon 2005]. A parent location step can also optionally be specified with ‘..’. The optional operator ‘..’ can be very useful to replace the *parent* axis in the XPath query for XML databases that do not support XPath axes.

Semantic used in semantic transformation is very straight forward by using information in Q . Take, for instance, the XPath query ‘dblp/*/title/parent::*’ which performs the task to query information of all the parents of **title**. By using Q , the unique path that selects the **title** is located first then any unique path that selects the parent of **title** and occurs before the unique path that selects the **title** is the semantic XPath query. This example will be revisited later on in great details by the proposed semantic transformation rules.

The semantic transformation rule called **ST_p**, which aims to remove the *parent* axis from an XPath query, is now proposed.

ST_p Rule. Semantic XPath query S is derived as follows:

- a. When ϵ is ‘*’ or node() and ∂ and ϑ are valid element names, then locate q_k that contains a ϑ as a target element if Q is used. Otherwise ϑ is a non-target element if S is used.

- b. When ε and ϑ are valid element names and ∂ is $*$ or $\text{node}()$, then locate q_k that contains ε as target element if Q is used. Otherwise ε is a non-target element if S is used.
- c. When ε and ϑ are $*$ or $\text{node}()$ and ∂ is a valid element name, then locate any q that contains ∂ as a target element. It then uses current q to locate q_k where q_k contains ϑ as the parent of ∂ .

$$S = \{q_i, \dots, q_k\} \text{ where } 0 \leq i \leq \kappa \text{ where each } q \text{ must contain } \vartheta \text{ or } \varepsilon$$

The following examples illustrate how S is derived.

$$p = */child::*/child::o/parent:* \quad Q = \{q_1, \dots, q_{12}, \dots, q_{17}, \dots, q_{19}, q_{20}, \dots, q_{22}\}$$

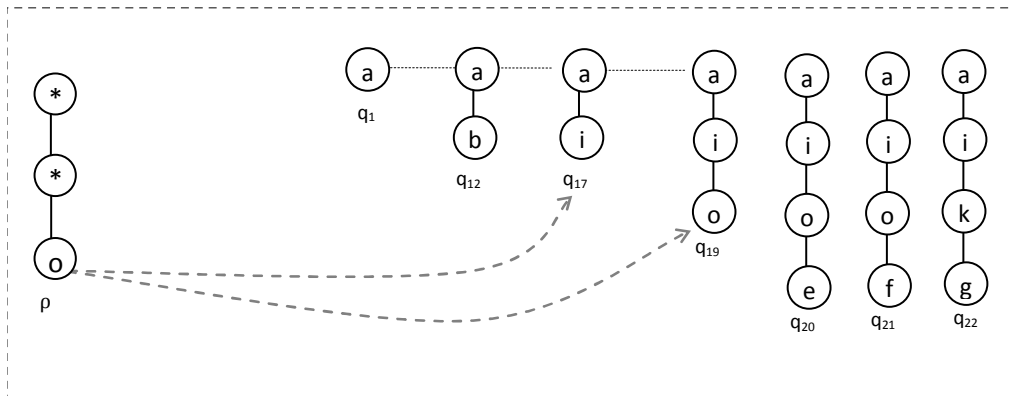


Figure 5.5 Semantic Transformation for Parent Axis Location Step

Following ST_p rules, rule c is suitable as it is based on Q and given p in Figure 5.5. ∂ is a valid element name and ε and ϑ are $*$. The value of ∂ is o . Due to the presence of the transforming location step $parent::*$ as a target location step in p , the transformation uses Q to first locate q where q has o as a target element. The unique path q_{19} has elements o as a target element. The unique path q_{19} has a target element o , which is also the parent element of element o in q_{19} . As a result, S is derived as follows:

$$S = \{q_{17}\}$$

Example. A requirement selects the parents of all item titles in the DBLP database.

XPath query $p = \text{dblp}/*/title/parent::*$

Based on given p , the values of the required parameters are now extracted as below

- β is ***parent::****
- ϵ is *******
- ∂ is ***title***
- ϑ is *******
- Q is {dblp, dblp/article, dblp/article/author, dblp/article/title,....,
dblp/inproceedings/title, dblp/proceedings/title,...., dblp/www/title,...}
- S is empty as the parent location step is the target location step.

Following **ST_p** Rules, rule c is identified for transforming p .

Since the ***parent*** axis is specified with ******* and ***title*** is the target element in p , ***title*** appears in more than one unique path in Q ; therefore the unique path that has a target element as the parent element of ***title*** will be selected. In this case, the transformation locates **dblp/article**, **dblp/inproceedings**, **dblp/proceedings**, **dblp/incollection**, **dblp/book**, **dblp/phdthesis**, **dblp/www**. Note that each of these unique paths produces its attributes or keys; hence, S is derived as follows:

$S = \{\text{dblp/article/@mdate, dblp/article/@key, dblp/inproceedings/@mdate,...,}$
 $\text{www/@mdate, www/@key}\}.$

Function **4. tranformParent** translates the semantic transformation rules for transforming an XPath query specified with the ***parent*** axis. The function first determines whether the parent element is a target element or a non-target element (Line 4:1).

When the transforming location step is a target location step, the transformation is straight forward to build the *tempList* by locating the unique path that has the parent element as a target element (Lines 4:2 – 4:7). If both context and parent elements are *******, a message ‘**Retain**’ will be assigned to the *tempList* and transformation is not required. This informs the main algorithm that the parent location step remains as it is (Line 4:7); therefore, no transformation is needed.

In the case where the transforming location is determined as a non-target location step, *semanticXPath* list is used. If both context and parent elements are *, then the *tempList* is a *semanticXPath* list and transformation is no longer required. When the context element is '*' and the parent element is a valid element name then it locates the unique path that has parent element. When the context is a valid element name and the parent element is *, it locates a unique path that has the parent of the context element and no context element is present (Lines 4:8-4:13). For example, the XPath query **dblp/*/title/parent::*/*author**. In this case, the transformation locates any unique path that has **author** as a target element and the parent of **author** should also be the parent of **title**. Hence

$S = \{ \text{dblp/article/author, dblp/inproceedings/author, dblp/proceedings/author, ..., dblp/www/author} \}.$

Function 4: List transformParent(String eContext, String parent, List semanticXPath)

```

    Let tempList be, empty list, tempQ be an empty list,  $\omega = \{*, \text{node}()\}$ , Q be the list of unique paths q
4:1  IF semanticXPath is empty THEN tempQ = Q ELSE tempQ = semanticXPath, semanticXPath = NULL
4:2  FOR each unique path q in tempQ
4:3    IF (((parent is in  $\omega$  && eContext element is valid) && context as target element in q) ||
4:4      ((eContext is in  $\omega$  && parent element is valid) && parent element in the second right most location step in q)) THEN
4:5      push q to tempList
4:6    ELSE IF (parent and eContext are in  $\omega$ ) Then tempList = 'Retain'; Exit
4:7  END LOOP
4:8  ELSE IF semanticXPath is not empty THEN
4:9    FOR each q in tempQ
4:10   IF (eContext is in  $\omega$  && parent is in  $\omega$ ) THEN tempList = semanticXPath; Exit
4:11   ELSE IF (eContext is in  $\omega$  && parent is valid) THEN locate q that has parent element and push q to tempList
4:12   ELSE IF (eContext is valid && parent is in  $\omega$ ) THEN locate q that has context element and push q to tempList
4:13  END LOOP
4:14  IF (tempList is empty) THEN semanticXPath = NULL
4:15  ELSE semanticXPath = tempList
4:16  RETURN semanticXPath

```

Once the *tempList* is completely built, it will be assigned to *semanticXPath* list (Line 4:15). If *tempList* is empty, it indicates a conflict has been detected (Line 4:14). *semanticXPath* is returned to the main algorithm (Line 4:16).

5.6 Semantic Transformation for Descendant or Descendant-or-self Axis

An optional operator ‘//’ has been recommended for the *descendant* axis [W3C, 1999; 2007a; 2010]. The semantic rule to transform XPath query location steps specified with a *descendant* or *descendant-or-self* axis is proposed.

The semantic transformation aims to remove a *descendant* or *descendant-or-self* axis from the XPath query. The expected semantic XPath query may result in multiple semantic XPath queries or a single semantic XPath query.

Before the semantic transformation rule is proposal, the XPath query ‘dblp/article/*/descendant::tn’ is now used to give a brief overview of how semantics are used in semantic transformation by eliminating XPath axis. This XPath query performs a task to select the descendant **tn** of article. By using list Q , the unique path that selects **tn** of article is first located. Any unique path which appears after the located unique path, and also selects **tn** in which the article element is also included in the unique path is the semantic XPath query. This example will be revisited later on in great details by the proposed semantic transformation rules.

For each descendant or descendant-or-self location step in an XPath query p , a set of semantic transformation rules called \mathbf{ST}_{dos} is now proposed.

\mathbf{ST}_{dos} . Semantic XPath query S is derived as follows:

- a. When ϵ is ‘*’ or node() and ϑ and ∂ are valid element names, then locate all in Q or S where each q selects a descendant of ∂ . q_k selects ∂ if α is *descendant-or-self*.
- b. When both ϵ and ∂ are valid element names and ϑ is ‘*’ or node(), then locate all q in Q or S where each q contains ∂ and selects ϵ if Q is used; or each q contains both ∂ and ϵ if S is used.
- c. When ϵ and ϑ are valid element names and ∂ is ‘*’ or node(), then locate all q in Q or S where each q contains ϑ and selects ϵ if Q is used; or each q contains both ϑ and ϵ if S is used.

- d. When ϵ is a valid element name and ϑ and ∂ are '*' or node(), then locate q_i to q_k in Q or S where each q select ϵ .

$S = \{q_i, \dots, q_k\}$ where $k = k-1$ if α is descendant axis and q_{k-1} does not select the context node itself

The following examples show how S is derived.

$$p = */child::i/child::o/descendant::* \quad Q = \{q_1, \dots, q_{12}, \dots, q_{17}, \dots, q_{19}, q_{20}, \dots, q_{22}\}$$

Following **ST_{dos}**, rule **a** is suitable based on given p , the value of ϵ is *, the value of ∂ is **o**.

The unique paths q_{19} , q_{20} and q_{21} in Figure 5.6 have elements **o** as the context element. The transformation needs to locate unique paths that have a valid context element **o**, which also has descendants.

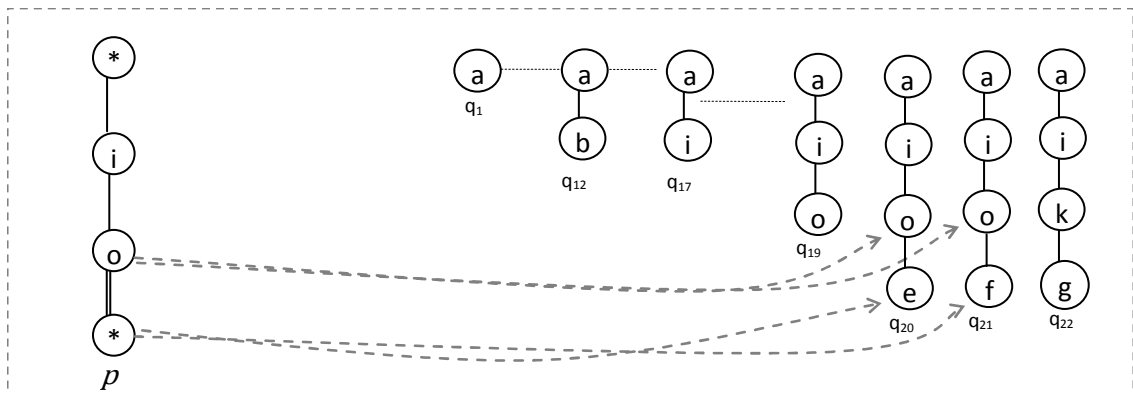


Figure 5.6 Semantic Transformation for Descendant or Descendant-or-self Axis Query

The mapping starts, with dot-line arrows, from element * in p to those expected descendants in q_{20} and q_{21} and element **o** in p to element **o** in q_{20} and q_{21} .

$$S = \{q_{20}, q_{21}\}$$

Example. A requirement selects all information of the titles, including the title name, of articles in the DBLP database.

XPath query $p = \text{dblp/article/title/descendant-or-self::}^*$

Based on given p , the values of the required parameters are extracted as follows:

- β is *descendant-or-self::**
- ∂ is **title**
- ϵ is *
- ϑ is **article**
- Q is {dblp, dblp/article, dblp/article/author, dblp/article/title, dblp/article/title/tt, dblp/article/title/tn, dblp/article/title/ref,...dblp/inproceedings/title, dblp/proceedings/title,..., dblp/www/title,...}
- S is empty as the descendant-or-self location step is a target location step in p

Based on ST_{dos} , rule **a** is suitable to transform p as ∂ and ϑ are valid, ϵ is '*'. Unique paths **dblp/article/title**, **dblp/article/title/tt**, **dblp/article/title/tn** and **dblp/article/title/ref** are selected.

In this example, the semantic XPath queries based on ST_{dos} would be produced as follows:

$S = \{\text{dblp/article/title}, \text{dblp/article/title/tt}, \text{dblp/article/title/tn}, \text{dblp/article/title/ref}\}$

Since unique path **dblp/article/title** does not have any key or attribute, selecting **title** would produce nothing for self information.

Let us consider another example using the descendant axis that is specified with an element which does not have descendants.

Example. A requirement selects all the information under the title names in the DBLP database.

XPath query $p = \text{dblp/article/*/descendant::tn}$

Based on given p , the values of required parameters are extracted as follows:

- β is *descendant::tn*

- ∂ is *
- ϵ is **tn**
- ϑ is **article**
- Q is {dblp, dblp/article, dblp/article/author, dblp/article/title, dblp/article/title/tt, dblp/article/title/tn, dblp/article/title/ref,...,dblp/inproceedings/title, dblp/proceedings/title,..., dblp/www/title,...}
- S is empty

Following **ST_{dos}**, rule **c** is suitable as ϵ and ϑ have valid element names and ∂ is *. The rule locates all unique paths that have **article** as a non-target element with **tn** as a descendant element. As a result, unique path {dblp/article/title/tn} is located. Since **tn** does not have any descendants, S is derived as follows:

$$S = \{\text{dblp/article/title/tn}\}$$

Function **5 transformDescendants** translates semantic transformation rules for transforming an XPath query that is specified with *descendant* or *descendant-or-self* axis. When the transforming location step is specified with *descendant-or-self* axis, *descendantOnly* is set to false. The function **transformDescendants** checks to determine whether the transforming location step is a target or non-target location step (Line 5:1). When the transforming location step is a target location step, the transformation builds *tempList* by locating q given a valid descendant element and the context element is * or, context and descendant elements are valid or both descendant and context elements are '*' (Lines 5:2 – 5:9).

When the transforming location step is a non-target location step *tempList* is built by selecting q given a valid descendant element and the context element is '*' or, valid context and descendant element names or both context and descendant elements are '*'.

(Lines 5:10 – 5:15). The *tempList* now contains a set of unique paths that produces information about descendants only.

Before the transformation verifies the axis for *descendant* or *descendant-or-self*, it must check a termination condition, that is, that the *tempList* is empty (Line 5:16).

Note that function **5** has a Boolean type parameter *descendantOnly*, which is used to determine that the axis is *descendant* or *descendant-or-self*. If the *descendantOnly* flag is false, the transformation transforms a location step specified with *descendant-or-self* axis, an addition *q* that selects the descendant itself to the list (Lines 5:17-5:24). The *tempList* is assigned to *semanticXPath* list and returns to the main algorithm (Lines 5:25-5:26).

Function 5: List transformDescendants(String eContext, String descendant, List semanticXPath, Boolean descendantOnly)

```

    Let TempQ,tempList be an empty lists,  $\omega = \{*, node()\}$ , Q be the list of unique paths q
5:1  IF semanticXPath is empty THEN tempQ = Q Else tempQ = semanticXPath, semanticXPath=NULL
5:2  IF semanticXPath is empty
5:3    FOR each unique path q in tempQ
5:4      IF (((eContext is in  $\omega$  && descendant is valid) && q has descendant as target element) ||
          ((eContext is valid && descendant is valid) && q has descendant as target element and eContext
            as non-target)) THEN
5:5        push q to tempList
5:6      ELSE IF (eContext is in  $\omega$  && descendant is in  $\omega$ ) Then
5:7        Locate first valid element  $\omega$  && eContext element determined based on its hierarchy in p.
5:8        Locate q that has determine eContext as non-target element && push q to tempList
5:9      End Loop
5:10     ELSE IF semanticXPath is not empty Then
5:11       FOR each q in tempQ
5:12         IF (((eContext is in  $\omega$  && descendant is valid) && q has descendant as non target element) ||
            ((eContext is valid && descendant is valid) && q has descendant and eContext as non-target
              elements)) Then
5:13           push q to tempList
5:14         ELSE IF (eContext is in  $\omega$  && descendant is in  $\omega$ ) THEN tempList = tempQ; Exist
5:15       End Loop
5:16     IF tempList is empty THEN semanticXPath=NULL; EXIST
5:17     ELSE IF (tempList not empty && descendantOnly is false) THEN
5:18       FOR each q in tempList
5:19         Obtain eContext element  $\tau$  in q
5:20       End Loop
5:21       FOR each q in unique path Q
5:22         Find q with  $\tau$  and attribute/id/key as target element
5:23         q to tempList
5:24       End Loop
5:25     ELSE tempList is not NULL semanticXPath = tempList
5:26     RETURN semanticXPath

```

5.7 Semantic Transformation for Self Axis

The *Self* axis navigates information of a context element. It has an optional ‘.’. For example, an XPath query is as follows:

dblp/article/self::* is equivalent to **dblp/article** or **dblp/article/.**

The semantic transformation for XPath query specified with the *self* axis to obtain an equivalent XPath query is now proposed. In the proposed semantic transformation, the goal is to remove these axes by using unique paths Q .

Wildcard expression ‘*’ used with an axis such as the *self* axis in an XPath query has been identified as one of the causes of query performance issues. Works have been performed to study the minimization of the tree patterns or containment that include ‘*’ without using semantics [Amer-Yahia et al., 2001; et al., 2002; Furfaro & Masciari, 2003]. Here, a different technique is proposed. That is a semantic transformation to achieve similar goals.

First let us address the semantic transformation for the *self* axis. The semantic transformation rule called ST_s is now proposed.

ST_s Rule. Semantic XPath query S is derived the following rules are satisfied.

- a. When ϵ is * or node() and ∂ is a valid element name, then locate $\{q_i, \dots, q_k\}$ in Q or S where each q contains ∂
- b. When ϵ and ∂ are valid element names, then locate $\{q_i, \dots, q_k\}$ in Q or S where each q contain both ϵ and ∂ as a target or non-target element depending on whether Q or S is used, ∂ and ϵ are the same element.
- c. When ϵ and ∂ are * or node() then p will be processed to get the next valid element e . Locate $\{q_i, \dots, q_k\}$ where each q contains e

$$S = \{q_i, \dots, q_k\} \text{ where } 0 \leq i \leq k$$

Figure 5.7 show how S is derived for the *self* axis specified in a location step.

Following $\mathbf{ST_s}$, rule **a** is suitable to transform p as ε is $*$ and ∂ is a valid element name in the given p .

By mapping elements in p to elements in q of Q using arrow-dotted lines in Figure 5.7, unique path q_{19} has elements o as ∂ and ε is $*$ which is represented by o that satisfies the requirement of XPath query. As the result, the transformation derives S as below:

$$S = \{q_{19}\}$$

$$p = */child::i/child::o/self::*$$

$$Q = \{q_1, \dots, q_{12}, \dots, q_{17}, \dots, q_{19}, q_{20}, \dots, q_{22}\}$$

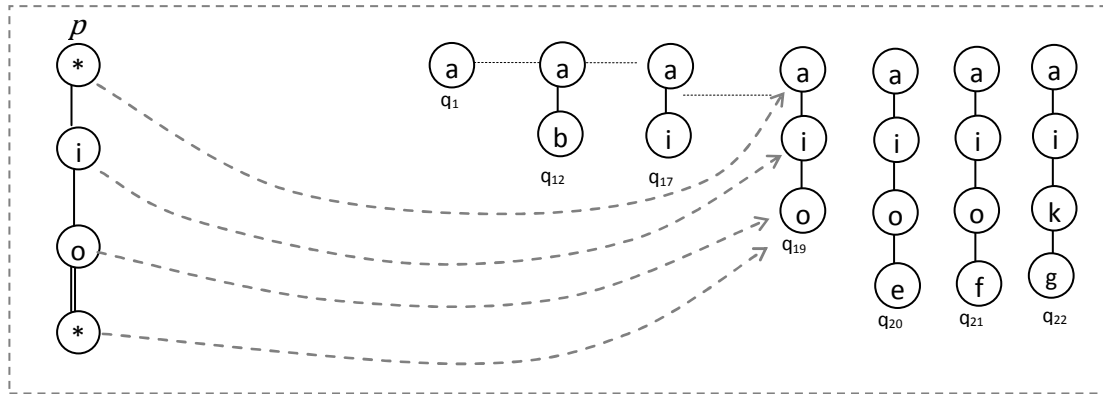


Figure 5.7 Semantic Transformation for *Self* Axis Query

Let us consider an example based on the DBLP schema that produced a list of derived unique paths Q .

Example. A requirement selects the titles of articles in the DBLP database.

XPath Query $p = \text{dblp/article/title/self}::*$

p is specified with the *self* axis; we refer readers to the previous example for the values of parameters β , ε , ∂ , Q , and S

Following $\mathbf{ST_s}$, rule **a** is suitable. The semantic transformation locates all unique paths that have article title as a target element, which is **dblp/article/title**. As a result, S is derived as follows:

$$S = \{\text{dblp/article/title}\}$$

Function **6. transformSelfAxis** translates the semantic transformation rules for transforming XPath query specified with the *self* axis.

The function first checks to confirm whether the *self* location step is a target location step or a non-target location step (Line 6:2).

Function 6: List transformSelfAxis (String eContext, String Self, List semanticXPath)

```

    Let tempList be, empty list, tempList and tempQ be empty lists,  $\omega = \{*, node()\}$ , Q be the list of unique paths q
6:1  IF semanticXPath is empty THEN tempQ = Q ELSE tempQ = semanticXPath, semanticXPath = NULL
6:2    FOR each unique path q in tempQ
6:3      IF (self is in  $\omega$  or node()) && eContext is valid and q has eContext as a target element) THEN
6:4        push q to tempList
6:5      ELSE IF (self is valid && eContext is in  $\omega$  or node()) THEN
6:6        push q to tempList IF q has self as target element
6:7      ELSE IF (self && eContext are in  $\omega$ ) THEN
6:8        process next valid element in p && locate q using hierarchy of self in p && push q to tempList
6:9    END LOOP
6:10 ELSE IF semanticXPath is not empty THEN
6:11   FOR each q in tempQ
6:12     IF ((self is in  $\omega$  && eContext is target element in q) ||
        (self is a valid && eContext is in  $\omega$ )) THEN
6:13       push q to tempList IF q has eContext or self as a non-target element
6:14     ELSE IF (self && eContext are in  $\omega$ ) THEN push q to tempList
6:15   END LOOP
6:16 IF (tempList not empty) THEN semanticXPath = tempList
6:17 ELSE IF (tempList is empty) THEN semanticXPath = NULL
6:18 RETURN semanticXPath

```

When the transforming location step is a target location step, the transformation uses *Q* to derive *tempList*. *q* is located to build the *tempList* given *self* specified with “*” and valid context element or, *self* is a valid element name and the context element is *. When both *self* and context elements are specified with “*”, *p* is used to locate first valid element then the new *q* is located that has the first located valid element as a non-target element (Lines 6:1 - 6:9).

When the transforming location step is a non-target location step, the transformation uses *S* to derive *tempList*. *q* is located to build the *tempList* given *self* specified with “*” and a valid context element name, select *q* that has the context element as a non-target element or; *self* is specified with a valid element name and the context element is “*”. When both *child* and context elements are “*”, *p* is used to locate the first valid

element e then new q is located that has first located valid element as non-target (Lines 6:10- 6:16).

After transformation, if the *tempList* is empty, this means that no semantic XPath query has been produced; this is caused by many factors, one of which is that semantic conflicts have been detected in the XPah query (Line 6:17). The transformation produces a list of semantic XPath queries that may contain a single semantic XPath query or a set of queries (Line 6:18).

5.8 Semantic Transformation for Child Axis

The *child* axis in a location step of an XPath query can be optionally specified. For example, **dblp/article/title/child::*** is equivalent to **dblp/article/title/***

The semantic transformation for the *child* axis to obtain the equivalent XPath query is now proposed. It aims to remove these axes by using unique paths Q . In the process of elimination of the *child* axis, semantic transformation may produce multiple semantic XPath queries if the context element has more than one child in the schema. Take, for instance, the XPath query ‘dblp/article/crossref/child::*’ means to select all the children of **crossref** of article. By using information in Q , the unique path that selects **crossref** of article is first located. Any unique path appears after the located unique path but also includes the **crossref** of article as the parent of the target element (right most element) is selected as semantic XPath query.

A wildcard expression used with the *child* axis in an XPath query has been identified as one of the causes of query performance issues. Works have been performed to study the minimization of the tree patterns or containment that include ‘*’ without using semantics [Amer-Yahia et al., 2001; et al., 2002, Furfaro & Masciari, 2003]. Here a semantic transformation is proposed to achieve similar goals.

First, let us address the semantic transformation for the *child* axis. The semantic transformation rule called **ST_c** is now proposed.

ST_c Rule. Semantic XPath query S is derived as follows:

- a. When ϵ is $*$ or $\text{node}()$ and ∂ is a valid element name, then locate $\{q_i, \dots, q_k\}$ in Q or S where each q contains ∂
- b. When ϵ and ∂ are valid element names, then locate $\{q_i, \dots, q_k\}$ in Q or S where each q contains both ϵ and ∂ . ϵ is a target or non-target element depending on whether Q or S is used and ∂ is the parent of ϵ .
- c. When ϵ and ∂ are $*$ or $\text{node}()$ then p will be processed to obtain the next valid element. Locate $\{q_i, \dots, q_k\}$ where each q the next valid element.

$$S = \{q_i, \dots, q_k\} \text{ where } 0 \leq i \leq k$$

The following examples are used to show how S is derived for the *child* axis specified in a location step

$$p = */\text{child}::i/\text{child}::o/\text{child}::* \quad Q = \{q_1, \dots, q_{12}, q_{17}, q_{19}, q_{20}, \dots, q_{22}\}$$

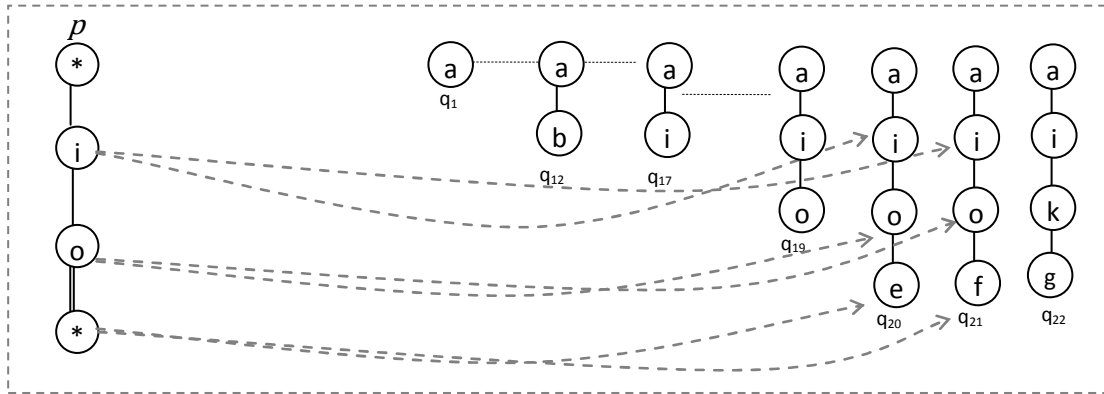


Figure 5.8 Semantic Transformation for Child Axis Query

Following ST_c , rule **a** is suitable as ϵ is $*$ and ∂ is a valid element name in p .

By mapping elements in p to elements in q of Q using arrow-dotted lines in Figure 5.8, unique paths q_{20} and q_{21} have elements o as ∂ and ϵ are e and f that satisfy the requirement of the XPath query as they demonstrate the children e and f of context elements o . As a result, the transformation derives S as follows:

$$S = \{q_{20}, q_{21}\}$$

Let us consider an example based on the DBLP schema that produced a list of derived unique paths Q .

Example. A requirement selects all the children of context element, namely the cross reference for articles in the DBLP database.

XPath query $p = \text{dblp/article/crossref/child::}^*$

The values of the required parameters from p are extracted as shown below:

- β is ***child::****
- ε is *****
- ∂ is **crossref**
- Q is {dblp, dblp/article/crossref, dblp/article/crossref/href, dblp/article/book/title,...}
- S is empty

Following **ST_c**, rule **a** is suitable. The semantic rule first locates all unique paths q that have article **crossref** as a non-target element and an immediate child of **crossref** as target element. As a result, the transformation locates **dblp/article/crossref/href**. Hence,

$$S = \{\text{dblp/article/crossref/href}\}$$

Function **7 TransformChildAxis** translates the semantic rules for transforming an XPath query specified with the ***child*** axis.

The transformation first determines whether the transforming location step is a target location step in the XPath query (Line 7:1).

When the transforming location step is a target location step, the transformation uses Q to derive *tempList*; q is located to build the *tempList* given *child* specified with ‘*’ and a valid context element name in p . Select q if q has a valid context element name as a non-target element or *child* is specified with a valid element name and context element is ‘*’. Otherwise, select q if q has a child axis specified with a valid target element. If both *child* and the context elements are specified with *, obtain the first valid element in p , and then locate q that has the obtained valid element as a non-target element (Lines 7:2-7:9).

When the transforming location step is a non-target location step, the transformation uses S to derive $tempList$, q is located to build the $tempList$ given $child$ is specified with ‘*’ and a valid context element name, or $child$ is specified with a valid element name and context element is ‘*’. When both $child$ and the context elements are specified with ‘*’, use p to locate the first valid element and then locate q that has the first located valid element as a non-target element (Lines 7:10-7:15).

Function 7: List transformChildAxis (String eContext, String Child, List semanticXPath)

```

    Let tempList be an empty list, tempList and tempQ be empty lists,  $\omega = \{*, node()\}$ , Q be the list of unique paths q
7:1  IF semanticXPath is empty THEN tempQ = Q Else tempQ = semanticXPath, semanticXPath = NULL
7:2  FOR each unique path q in temp
7:3    IF (child is in  $\omega$  && eContext in q has immediate child as target element) THEN
7:4      push q to tempList
7:5    ELSE IF (child is a valid && eContext is in  $\omega$ ) THEN
7:6      push q to tempList IF q has child as target element
7:7    ELSE IF (child && eContext are in  $\omega$ ) THEN
7:8      process next valid element in p && locate it in q then push q to tempList
7:9    End Loop
7:10 ELSE IF semanticXPath is not empty THEN
7:11   FOR each q in tempQ
7:12     IF ((child is in  $\omega$  && eContext exist in q)|| (child is a valid element && eContext is in  $\omega$ )) Then
7:13       push q to tempList IF q has eContext as a non-target element
7:14     ELSE IF (child && eContext are in  $\omega$ ) Then push q to tempList
7:15   END LOOP
7:16 IF (tempList not empty) THEN semanticXPath = tempList
7:17 RETURN semanticXPath

```

When the $tempList$ is empty, it means that semantic conflicts have been detected in the XPath query (Line 7:16). Otherwise, a list of semantic XPath queries may contain a single semantic XPath query or a set of semantic XPath queries (Line 7:17).

5.9 Conclusion

The semantic transformation rules and their algorithms for XPath axes including *{following-sibling, preceding-sibling, following, preceding, ancestor, ancestor-or-self, parent, descendant, descendant-or-self, self, child}* have been proposed.

For axes such as *parent, child, descendant, descendant-or-self, ancestor, ancestor-or-self and self*, the transformations are straightforward as paths are traversed in a

vertical direction. On the other hand, for the remaining axes - *following*, *preceding*, *following-* and *preceding-sibling* - the transformation needs to combine both vertical and horizontal directions for traversing paths.

The proposed semantic transformation rules have also been translated to algorithms. Each algorithm provides a feature to detect the unsatisfied query. These algorithms are then implemented for our performance evaluation discussed in Chapters 8 and 9.

Chapter 6

Semantic Transformations for XPath Queries Specified with Predicates

This chapter proposes the third category of semantic transformations to transform XPath queries specified with predicates.

A predicate in an XPath query expresses a query condition to be fulfilled. The query condition is a Boolean expression. It may involve comparisons of elements and values, path expressions denoting elements to be compared as well as further path expressions. The proposed semantic transformations for XPath queries specified with predicates enable predicates to be removed or modified for optimization purpose. Among the benefits of semantic transformations is one that detects and removes any redundancy, which could be the whole predicate, in the query that may impact upon performance. Otherwise, if predicates are retained, the semantics also show how they can be semantically transformed to boost efficiency and reduce resource utilization.

As in chapters 4 and 5, the semantic transformations in this chapter use information lists Q and C . Unique paths in list Q and constraints of elements in list C have been proposed to be derived from XML Schema [W3C, 2004a; 2004b] in Section 4.1

chapter 4, are used to propose the transformations. A summary is now presented to remind the readers about the derivation of Q and C . List Q contains a list of unique paths q where each q is a sequence of elements that express from root to a particular element in a given XML Schema. Only the parent-child “/” relationship among the elements is in q . List C contains a list of c where each c contains the element and its constraints specified in the XML Schema.

6.1 Semantic Transformation for Predicates: Motivation

Each XPath query consists of a sequence of location steps (as described in chapter 3). Chapter 3 also described predicates as an optional component in an XPath query; however, its presence plays a critical role in filtering required information.

A query condition may involve comparison between elements and values, path expressions denoting elements and further path expressions [Diao et al., 2003; Wu, et al., 2003]. The complexity of XPath predicates is based on different types of query condition connections (hence, OR/AND) and types of query condition comparisons. Such complexity certainly produces a number of challenges, among these challenges is the query processing techniques and query performance. As part of this research, we propose a typology of semantic transformations for XPath queries specified with predicates for optimization purposes.

Figure 6.1 shows an overview of the proposed tasks in this chapter. The semantic transformations for predicates are divided into two proposed methods, which are *predicate elimination* and *predicate reduction semantic transformations*, shown as **(2a & b)**. In order to remove or modify the predicates in a given query for transformation, the query conditions in the predicates must be determined with a status, shown as **(1)**, that allows a necessary action to be taken; this is done by *condition status determination and algorithmic functions*, shown as **(3)**.

The semantic transformations to eliminate or reduce them can be processed if the predicates are given a valid status. If a semantic conflict is detected in a predicate; an empty result set will be returned and the transformation can be terminated without further processing.

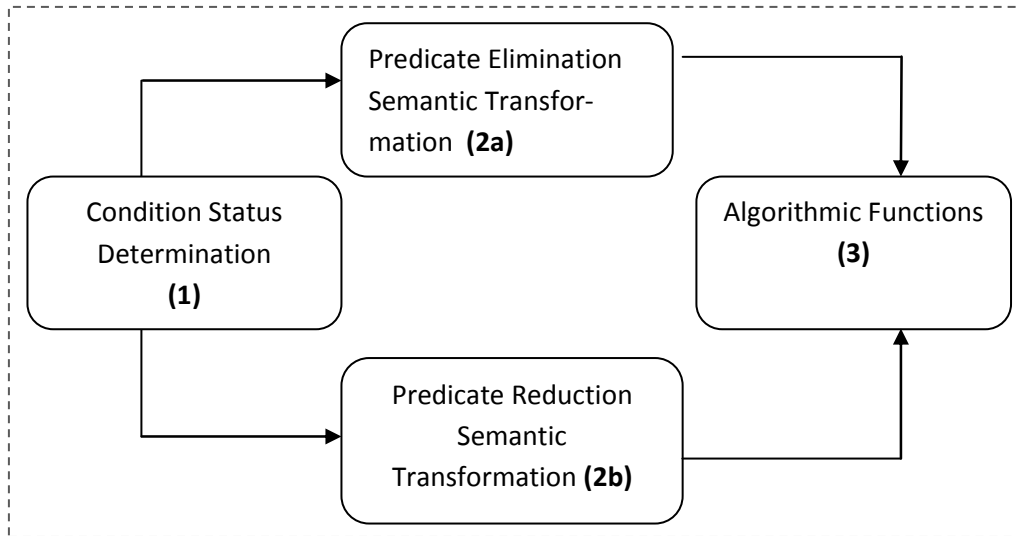


Figure 6.1 Semantic Transformations for XPath Queries Specified with Predicates Framework

6.2 XPath Query Predicate Structures and Global Parameters

This section addresses the structures of XPath query predicates and a common set of required parameters.

6.2.1 XPath Query Predicate Structures

This section addresses the XPath query predicate structure that consists of XPath query conditions that are expressed in an XPath query predicate.

A user-defined XPath query may be specified with a predicate that holds one or more query conditions to filter selected information. For example, consider the XPath query given below:

$$p = i//k/j[l = 3 \text{ or } m/n = li@id \text{ and } 2]/r$$

In p , XPath query predicate is $[l=3 \text{ or } m/n = li@id \text{ and } 2]$, where j is a branching element. The XPath query predicate in p consists of three query conditions including $l=3$, $m/n = li@id$ and 2. The first query condition $l=3$ is a comparison between element l and constant value 3. The second query condition $m/n = li@id$ is the

comparison between the path fragment m/n and a further path fragment value $/i@id$. The third query condition 2 is the evaluation of an index ordered value 2 of element j . The comparison values of three conditions are 3, $/i@id$ and 2 respectively.

6.2.2 Defining Global Parameters

Similar to chapter 5, the semantic transformations proposed in this chapter utilize the same terms and information related to XPath query, lists of Q and C to progress to semantic rules. This section defines a set of global parameters that reference to the terms and information which are used to define the all the semantic transformation rules in this chapter.

- e is a branching element where a query predicate occurs
- Pr is a query predicate that holds one or more query conditions
- σ is a query condition status, which is further extended in details in section 6.3.1
- ω is a query condition awarded with σ
- ϕ is a connective operator AND/OR
- γ is an XPath query comparison element/path fragment
- θ is a comparison value of γ
- S is a semantic XPath query

6.3 Semantic Transformations for Predicates

This section proposes for predicate elimination and predicate reduction semantic transformations. In order to modify or eliminate the predicate in a given XPath query, a methodology is now proposed to determine the status of the query conditions and their predicates.

6.3.1 Determination of Status for Query Conditions and Their Predicate

Before a function is defined to determine the status of query conditions specified in a predicate, one of the following three terms is adopted for referring to a status of a

query condition that is going to be determined by the function proposed here. The terms are defined as followed:

Definition 6.1 (Full-qualifier (F_Q)). Full-qualifier is a query condition status σ , which is determined for a query condition ω if and only if the comparison value θ of comparison element/path γ in ω matches with θ of the same element/path fragment γ located in Q and C .

For example, the XPath query ‘//phdthesis[supervisor = ‘dblp/phdthesis/@key’]/title/tn’ performs the task to find all the thesis titles that must have at least one supervisor who must also hold a PhD qualification. In this example the query condition can be associated with a full-qualifier status if in the given XML Schema, the **supervisor** must be referenced to an existing PhD thesis. This example will be revisited later in great details by the semantic transformation rule.

Definition 6.2 (Partial-qualifier (P_Q)). Partial-qualifier is a query condition status σ , which is determined for the query condition ω if and only if the comparison value θ of comparison element/path γ in ω matches some θ , but not all, of same γ located in Q and C .

For example, the XPath query ‘dblp/phdthesis[year =1999 or urls]’ performs the task of retrieving all the thesis that are published in year 1999 or have a url for viewing. In this example the query condition can be associated with a partial-qualifier status if, in the given XML Schema, the published year for a thesis is between 1995 and 2020. This example will be revisited later in great details by the semantic transformation rule.

Definition 6.3 (Conflict-qualifier). Conflict-qualifier is a query condition status σ , which is determined for the query condition ω when the comparison value θ of comparison element/path γ in ω does not match θ of the same γ located in Q and C .

For example, the XPath query ‘dblp/phdthesis[year =1995 or urls]’ performs the task of retrieving all theses that are published in year 1999 or have a url for viewing. In this example the query condition ‘urls’ can be associated with a conflict-qualifier

status if, in the given XML Schema, phdthesis node does not have any child labelled as 'urls'. This example will be revisited later in great details by the semantic transformation rule.

The examples above will be revisited after a function namely **conditionStatus**, is defined to determine the status, using one of the terms defined above, of the query conditions specified in a predicate.

$$\begin{array}{c} (Q, C) \\ \downarrow \\ \text{conditionStatus}(ePr) \rightarrow eR \end{array}$$

Pr is a predicate that contains a set of query conditions ω where $Pr = [\omega_{1(\Phi)} \omega_{2(\Phi)} \dots_{(\Phi)} \omega_n]$, and O is the logical connective operator representing AND/OR; e is a branching element. Q is a list of unique paths, C is a list of constraints of elements. Both Q and C are derived in chapter 4. R is the result based on the status of query conditions in Pr determined by the **conditionStatus** function.

ePr is the combined information of a branching element e and predicate Pr which consists of information that comes from the XPath query p . Each query condition ω in predicate Pr is evaluated based on information in Q and C . Each query condition is associated with *full-qualifier*, *partial-qualifier* or *conflict-qualifier*.

On obtaining the query condition status, the **conditionStatus** function takes into account the connective operator such as OR or AND or both so that R finally can be *full-qualifier*, *partial-qualifier* or *conflict-qualifier*. The comparison operators such as $\{!=, =, \leq, \geq, <, >\}$ used when XPath query comparison elements/path fragment have XPath query comparison values. ω could be a set if there exist one or more OR or AND or both where each comparison element/path γ in query condition ω is optionally specified with comparison value θ .

An XPath query $p = //phdthesis[supervisor = '/dblp/phdthesis/@id']/title/tn$. In this example, the predicate $R = [supervisor = '/dblp/phdthesis/@id']$ where its status needs to be identified as a *full-qualifier*, *conflict-qualifier* or *partial-qualifier*.

As shown in p , predicate R contains query condition is a comparison of the value of the **supervisor**; element with an absolute path **/dblp/phdthesis/@id**. By applying the function **conditionStatus**, it first verifies if **//phdthesis/supervisor** is a subset of a unique path in list Q . Second, the function also verifies whether **supervisor** is referenced to **dblp/phdthesis/@id** in constraint list C . As the result, function **conditionStatus** finds that **//phdthesis/supervisor** is a subset of the unique path **dblp/phdthesis/supervisor** in list Q and *supervisor* refers to id of an existing **phdthesis**. As a result, it associates status of *full-qualifier* to the query condition and ultimately to predicate R .

Definition 6.4 (Context Position Index) Context Position Index is an index position value of a context element, which is currently being processed for a return of index node in the sequence of nodes.

For example, considering an XPath query **dblp/book[2]** where 2 is a context position index on the context element **book**. The XPath query returns the second book as the result.

Context position index in a query form context is used when there is a query condition with no comparison element and enclosed with []. It is normally placed on the context element in the XML query.

6.3.2 Predicate Elimination Semantic Transformation

A predicate can be completely eliminated when it satisfies a set of semantic rules which are going to be proposed in this section. As in the previous section, we describe the derivation of the query condition status for query conditions specified in a predicate. A predicate may contain a single condition or multiple conditions.

A single query condition can be a value such as the position context index on a context node. A condition comparison is between an element/path fragment and a value or an existence of an element/path fragment; that is when a condition element is specified with no value.

Multiple query conditions can be connected using AND and OR connective operators between the pair of query conditions and each query condition can be specified with or without a comparison value. A query condition without a comparison value occurs when the query condition is specified with an element or a path fragment or a position index value on a context element.

For instance, consider the following queries: `//book[author]`, `//book[author/name]`, `//book/author[2]`, `//phdthesis[supervisor=//phdthesis/@id]`. The first query condition **author** from `[author]` is an element. The second query condition **author/name** from `[author/name]` is a path fragment. The third query condition 2 from `[2]` is a position index value that evaluates to return the second author from the sequence of authors. The fourth query condition **supervisor=//phdthesis/@id** from `[supervisor=//phdthesis/@id]` is a comparison element **supervisor** that has a value of the PhD thesis identification.

Here a set of semantic transformation rules is proposed that eliminates predicates from a given XPath query. When a predicate contains a set of query conditions, each of them must be given a full-qualifier (F_Q) or partial-qualifier (P_Q) by the functions in Section 6.3.1. The query conditions can be removed if they satisfy the set of semantic rules, namely ST_{pe} .

As mentioned earlier, a predicate in an XPath query expresses a query condition which is a Boolean expression. The logic of semantic rules ST_{pe} below are presented to remove a predicate when one of the three circumstances is satisfied. (a) When there is no connective operator present and the query condition is associated with a full-qualifier status whose comparison element has occurrence constraint of 1 or above. (b) When only connective OR is present among the query conditions which project the same comparison element. E.g. `[profession = 'lecturer' or profession = 'tutor']` is determined to $[P_Q \text{ or } P_Q]$ whose comparison element, profession, also has a constraint value of 1 or above. In the XML Schema, employee/profession is restricted with enumeration values of {lecturer, tutor}. (c) When only connective OR is presents among the query conditions, only one of the query conditions is associated with a full-qualifier status whose comparison element has a constraint value is 1 or above. E.g. `dblp[book/year >= 1995 or book/classification = 'classic']`

is determined to $[F_Q$ or $C_Q]$. This is because in the XML Schema a book restricted with a published year between 1995 and 2020, the classification of books is in several areas such as classic, action, romance, and many more.

The statement above is now put into a rule set of **ST_{pe} Rule** below. The aim of the rule set **ST_{pe} Rule** is to avoid the processing of unnecessary predicates in the XPath query to achieve optimization objectives. XPath query Predicate Pr is eliminated if one of the following rules is satisfied:

- a. When predicate Pr contains query condition whose associated status σ is F_Q , connective ϕ is NULL and comparison elements γ have minimal occurrences greater than 0.
- b. When predicate Pr contains query conditions ω whose associated status σ is P_Q , connective ϕ is only OR, comparison elements γ project the same element and their comparison values θ match all restricted values θ of the same γ located in C whose minimal occurrence is greater than 0.
- c. When Pr contains query conditions ω where one or more associated status σ is F_Q , connective ϕ is OR, comparison element of γ whose minimal occurrence is greater than 0.

An exceptional case arises when all query conditions are joined by conjunction operators and one of them is associated with a status of C_Q , this causes the whole predicate conflict and XPath query certainly returns an empty result. Hence XPath query is not needed to send to database for validation to avoid unnecessary query processing. This is an add-on feature automatically built in these proposed semantic transformation. This also applies to the rule set in section 6.3.3.

The following examples in Figure 6.2 show how XPath queries specified with predicates can be transformed by eliminating predicates. Two XPath queries p_1 and p_2 and a set of unique paths Q are used for the demonstration. Both p_1 and p_2 are specified with query conditions that have no comparison values. Also, a connection operator such as AND is used between the pair of query conditions in p_2 .

The condition status function ePr is derived and the values of the required parameters are extracted based on XPath queries p_1 and p_2 are given in Figure 6.2. as follows

$$ePr = i[o/e], \gamma = o/e \quad (p_1)$$

$$ePr = i[k \text{ AND } o/e], \gamma = o/e, \phi \text{ is AND} \quad (p_2)$$

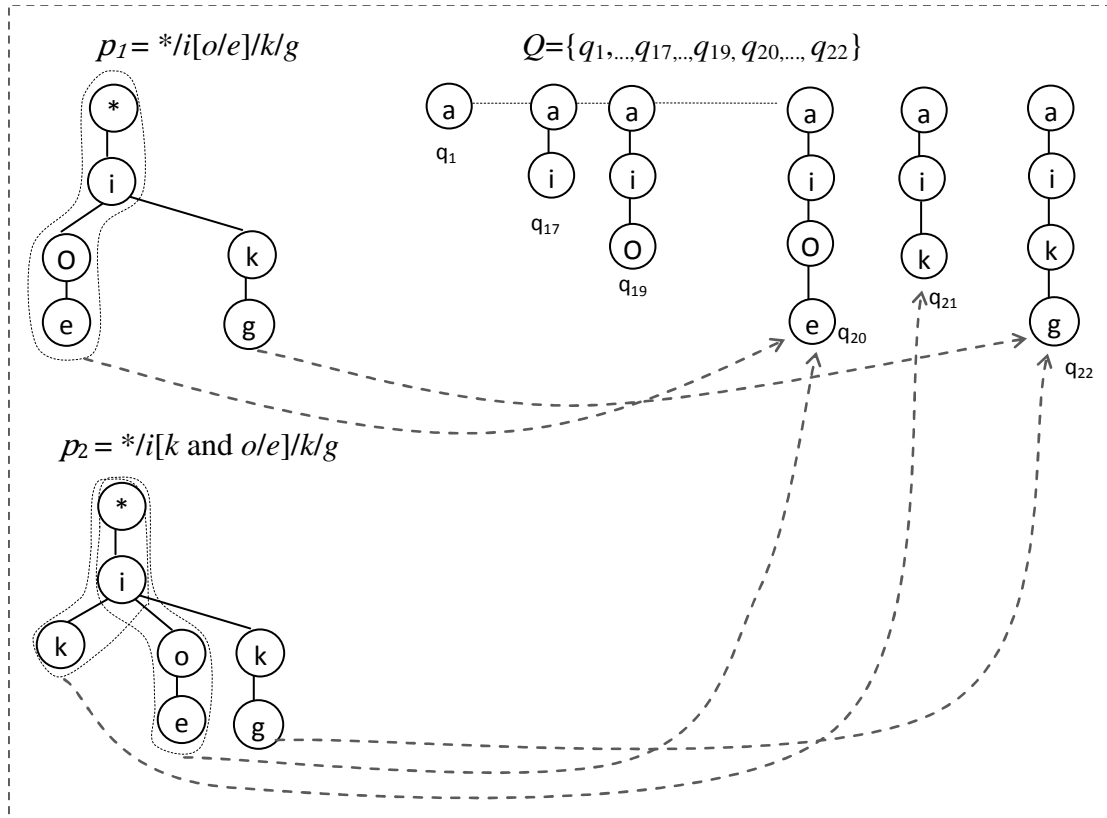


Figure 6.2 XPath Query Predicate Holds a Single XPath query Condition with No Value

As shown in Figure 6.2, p_1 has an enclosed dash-line around the path that expresses from the root specified by wildcard '*' to i then o and finally e in which o/e is the query condition specified in predicate $[]$. The elements, except the wildcard '*' operator, in the path within the enclosed dash-line are matched with those in the unique path q_{20} in Q as shown by an arrow dash-line from e in p_1 to e in q_{20} . As the wildcard operator '*' is an unidentified element in the path, it is matched to any eligible element that must be the ancestor of i in q_{20} . In this case it is the root element in q_{20} . The selected information is projected by the open path that traverses from the root specified with the wildcard * via k to g . The elements in the open path are

matched with those in a unique path q in Q as shown by an arrow dash-line from g in p_1 to g in unique path q_{22} . Both q_{20} and q_{22} are intersected at the branching element i .

As shown in Figure 6.2, p_2 has two enclosed dash-lines around the paths. The first enclosed dash-line shows a full path that contains the element k as the first query condition. The second enclosed dash-line shows a full path that contains the path fragment o/e as the second query condition. The elements, except the wildcard operator ‘*’, in the paths enclosed by dash-lines are matched with those unique paths q_{20} and q_{21} in Q as shown by an arrow dash-line from e in p_2 to e in q_{20} and k in p_2 to k in q_{21} . As the wildcard operator ‘*’ is an unidentified element in the path, it is matched to any eligible element that must be an ancestor of i in q_{20} and q_{21} . The selected information based on the path goes from the root to element g , which is mapped by an arrow dash-line from element g in p_2 to unique path q_{22} . Unique paths mapped for p_2 such as q_{20} , q_{21} and q_{22} are intersected at the branching element i .

The XPath query conditions in the XPath query predicates Pr of p_1 and p_2 first need to associate with the status by using function **conditionStatus** which has been proposed earlier.

The status of query conditions are obtained and shown in Figure 6.3. The **conditionStatus** function obtains status (as outputs) for predicates ePr (as inputs) in p_1 and p_2 (as XPath queries).

XPath Query	Predicate Inputs	Outputs
p_1	$ePr = i[o/e]$	$i[F_Q]$
p_2	$ePr = i[k \text{ AND } o/e]$	$i[F_Q \text{ AND } F_Q]$

Figure 6.3 Condition Status for Predicates in p_1 and p_2

To transform p_1 by following XPath query predicate elimination semantic transformation ST_{pe} , Rule **a** is used to remove the predicate $[o/e]$ from p_1 as the status σ satisfies a full-qualifier F_Q . In addition the comparison element o and e of γ are verified against the same set of elements in C , and they have minimal occurrences greater than 0.

To transform p_2 by following XPath query predicate elimination semantic transformation \mathbf{ST}_{pe} , Rule **c** is used to remove the predicate $[k \text{ and } o/e]$ from p_2 as the status σ satisfies a full-qualifier F_Q . In addition, the comparison element k , o and e of γ are verified against the same set of elements in C , and they have minimal occurrences greater than 0.

Below are the running examples to demonstrate the predicate elimination semantic transformation. The examples are based on DBLP XML Schema as shown in Appendix 1.

Example. Select all PhD theses of the candidates who must have a valid supervisor.

XPath query $p = //phdthesis[supervisor = '/dblp/phdthesis/@key']/title/tn$

As shown in p , XPath query predicate Pr is $[\text{supervisor} = '/dblp/phdthesis/@key']$ where branching element e is **phdthesis**.

First, function **conditionStatus** is used to determine the status of the XPath query condition $\text{supervisor} = '/dblp/phdthesis/@key'$ imposed on element **phdthesis**.

Figure 6.4 shows a key referenced constraint used for **supervisor** element in the DBLP Schema, which exists in list C and used by **conditionStatus** function.

```
<xs:selector xpath="dblp/phdthesis"/>
  <xs:field xpath="@key"/>
    <xs:key>
      <xs:keyref name="supervisor" refer="phdthesisKey">
        <xs:selector xpath="dblp/phdthesis"/>
        <xs:field xpath="supervisor"/>
      </xs:key>
    </xs:key>
```

Figure 6.4 Schema Fragment Which Exhibits a Keyref Constraint

After performing the function **conditionStatus**, it locates in C the following entries **phdthesis/supervisor=dblp/phdthesis/@key** and in Q the following entry **dblp/phdthesis/title/tn**. The located information determines the status result for query condition in predicate

$$ePr = \text{phdthesis}[F_Q].$$

Following semantic transformation rule **a** in **STpe**, the query condition is associated with full-qualifier and it is confirmed that query condition join operator is NULL in the predicate. Information entry “**phdthesis/supervisor occurrence 1 1**” is located in list *C*, which means the minimal occurrence of comparison path fragment **phdthesis/supervisor** is greater than 1 and its maximal occurrence is also 1. This also means for every existing **phdthesis**, there must be a valid **supervisor**. Hence the semantic transformation can now remove the predicate and produce semantic XPath query:

$$S = //\text{phdthesis}/\text{title}/\text{tn}$$

Another running example, as shown below, demonstrates XPath query specified with a predicate that has two query conditions. While the first query condition has a comparison element and a value, the second query condition has a comparison path fragment without any comparison value.

Example. Select all articles published in and after 1950 and which have a valid title.

$$\text{XPath query } p = \text{dblp}/\text{article}[\text{year} \geq 1950 \text{ and title}/\text{tn}]$$

In *p*, XPath query predicate *Pr* is [**year ≥ 1950 and title/tn**] where branching element *e* is **article**.

The function **ConditionStatus** is used to determine the status of the XPath query conditions **year ≥ 1950** and **title/tn** imposed on **article**.

After performing the function **conditionStatus**, the information entry **article/year inclusive 1950 2020** is located in *C* and entries **dblp/article/year** and **dblp/article/title/tn** are located in *Q*. The located information indicates that the first query condition **year ≥ 1950** is a semantic matching to **article/year inclusive 1950 2020** that shows articles published between 1950 and 2020 in the database. The second condition is the path fragment **title/tn** which is part of the unique path **dblp/article/title/tn**. Hence, the **ConditionStatus** function determines the status for predicate *Pr* such that:

$$ePr = \text{article}[F_Q \text{ and } F_Q]$$

By following semantic transformation rule **c** in **STpe**, both query conditions are associated with full-qualifier status that each query condition projects on different comparison elements such as **year** and **title\tn** respectively. The query conditions are joined by AND connective. The first query condition **year** has a comparison value 1950 and above, that matches the whole range of restricted values of **year** located in *C*. The second query condition **title\tn** has no comparison value. Information entries **article/year occurrence 1 ∞** and **article\tn occurrence 1 1** are located in list *C*. This means the comparison element **year** and comparison path fragment **title\tn** have minimal occurrences greater than 0. Hence, semantic transformation can now remove the predicate and produce the semantic XPath query as below

$$S = \text{dblp/article}$$

Another running example is now considered that an XPath query has query conditions joined by AND connective and both query conditions have no comparison values.

Example. Select all articles in which each article must have valid authors or title name.

$$\text{XPath query } p = \text{dblp/article}[\text{author or title}[\text{tn}]]$$

As shown in *p*, XPath query predicate *Pr* is **[author or title[tn]]** where branching element *e* is **article**.

First, function **conditionStatus** is used to determine the status of the XPath query conditions **author** and **title[tn]** imposed on **article**.

By using function **conditionStatus**, information entries **dblp/article/author** and **dblp/article/title\tn** are located in *Q*. The located information indicates that while first query condition path fragment **author** is a semantic match as part of unique path **dblp/article/author**, the second query condition path fragment **title[tn]** is a semantic match as part of unique path **dblp/article/title\tn**. Hence, the **conditionStatus** function determines the status for predicate *Pr* such that:

$$ePr = \text{article}[F_Q \text{ or } F_Q]$$

Following rule **c** in semantic transformation rules **STpe**, both query conditions have been associated with full-qualifier and projected on different comparison elements such as **author** and **title[tn]** respectively. The semantic rule **c** also confirms query conditions are joined by AND connective and locates entries **article/title/tn occurrence 1 1** and **article/author occurrence 1 ∞** in list *C*. As a result, comparison element **author** and comparison path fragment **title[tn]** have minimal occurrences greater than 0. Hence, the semantic transformation rule **c** in **STpe** can now remove the predicate and produces semantic XPath query as below

$$S = \text{dblp/article}$$

6.3.3 Predicate Reduction Semantic Transformation

Not all XPath query predicates can be eliminated during semantic transformations. However, the query conditions in the predicates can be modified. That is, some conditions can be transformed so that they can be removed to reduce the size of the predicates. This is referred to as *predicate reduction semantic transformation*. Similar to the *elimination semantic transformation*, the status of each query condition first needs to be determined before reduction semantic transformation can take place.

A predicate that has a single query condition cannot be reduced by semantic transformations. On the other hand, not all multiple XPath query conditions in an XPath query predicate can be modified. The query condition that has a conflict-qualifier status and is joined with another query condition may have a significant impact on the query result such that an empty or a non-empty result returned by the main query. The result depends on the type connective AND or OR that is used between the query conditions.

Definition 6.5 (Mutual Exclusive Condition). A mutual exclusive condition is a joined query condition that is connected with another query condition by an OR operator and whose status is a conflict-qualifier (C_Q) and its existence does not affect the result set returned by the main query.

A query condition that may have no impact on the result returned by the main query is referred to as a ‘mutual exclusive’ query condition.

To elaborate on the ‘mutual exclusive’ condition, consider an XPath query $p = \text{dblp/book}[\text{year} < 1950 \text{ or } \text{year} = 1952]$. C list contains entries where one of them is **book/year inclusive 1950 2020**. If Q has unique paths where one of them is **dblp/book/year** then branching node and predicate $ePr = \text{book}[\text{year} < 1950 \text{ or } \text{year} = 1952]$ is a fragment of **dblp/book/year**. The query condition **year < 1950** has a conflict-qualifier status because the comparison value is below the lower bound of 1950. The query condition **year = 1952** has a partial-qualifier status as it matches a value within the restricted bound. In this predicate, query condition **year < 1952** is a mutual exclusive query condition as its existence does not affect the result returned by the main query,

A mutual exclusive query condition that exists in the predicate is one of the query components that co-ordinate with the rest of query components within the query to make up the main component, that is the whole query. Hence it is also a co-existing query component.

The semantic rules \mathbf{ST}_{pr} below are presented to reduce the number of query conditions in a predicate when one of the three circumstances is satisfied.

(a) When connective OR is present among the query conditions, one or more query conditions are associated with conflict-qualifier status, which can be removed. For example in a simplified form of $\text{book}[\text{profession} = \text{'research'} \text{ AND } \text{year} = 1999 \text{ OR } \text{year} = 1952]$, the expression $\text{book}[P_Q \text{ AND } P_Q \text{ OR } C_Q]$ is equivalent to $\text{book}[P_Q \text{ AND } P_Q]$ if book has a published year restricted between 1995 and 2020 in the XML Schema. (b) When connective AND is present among the query conditions, if query conditions are associated with full-qualifier status and their comparison elements have constraint of 1 or above, then the query condition can be removed. For example the expression $\text{book}[\text{year} \Rightarrow 1995 \text{ AND } \text{profession} = \text{'lecture'}]$ is determined to $\text{book}[F_Q \text{ AND } P_Q]$ which is equivalent to $\text{book}[P_Q]$ if book has a published year restricted between 1995 and 2020. (c) When connective OR is present among the query conditions, which associate with partial-qualifier status and project the same condition element whose comparison element has a constraint value of 1 and above.

If the comparison values of the comparison element in these query conditions must satisfy a complete set of restriction values of the element in the schema then those query conditions can be removed. For example the expression $\text{phdthesis}[\text{supervisor/year} = 1999 \text{ AND } \text{profession} = \text{'teaching'} \text{ OR } \text{profession} = \text{'research'}]$ is determined to $\text{phdthesis}[P_Q \text{ AND } P_Q \text{ OR } P_Q]$. While in the XML Schema a supervisor thesis published is restricted between 1995 and 2010, profession is restricted with the enumeration values of {teaching, research}.

The statement above is now put into a rule set of **ST_{pr} Rule** below. The aim of the rule set **ST_{pr} Rule** is to avoid the processing of unnecessary predicates in an XPath query to achieve optimization objectives. Query conditions in XPath predicate Pr is eliminated if one of the following rules is satisfied:

- a. When predicate Pr contains query conditions ω and C_Q are status σ associated to some ω and connective ϕ is OR between the pair of those ω , then ω that has $\sigma = C_Q$ can be removed from P_r .
- b. When predicate Pr contains query conditions ω , P_Q and F_Q are status σ associated to ω . If connective ϕ is AND/OR or both between the pair of ω and comparison element γ of ω has minimal occurrences > 0 and ω is associated with F_Q , then ω can be removed from P_r .
- c. When predicate Pr contains query conditions ω , P_Q are status σ associated to ω . If connective ϕ is OR among ω that have $\sigma = P_Q$ which must project the same comparison elements γ and has a minimal occurrences > 0 , all comparison values θ of γ in those ω match all θ of γ in C then ω be removed from P_r .

The following example in Figure 6.5 shows how an XPath query predicate is transformed using lists Q and C .

The required parameters and their values are as follows

$$ePr = i[o/e = \omega_1 \text{ or } s], \gamma = \{o/e, s\}, o = \text{OR}, \theta = \{\omega_1\} \text{ where } \omega_1 \text{ is value of } e$$

As shown in Figure 6.5 p has two enclosed dash-lines. The first enclosed dash-line shows a path that contains the path fragment o/e as the first query condition in which

e has a comparison value ϖ_1 . The second enclosed dash-line shows a full path that contains the element s as the second query condition.

After performing the **conditionStatus**, the first query condition is associated with a P_Q status because the query condition o/e has value ϖ_1 which matches an exact value of same e in list C , which does not satisfy the range value as specified for o/e in C .

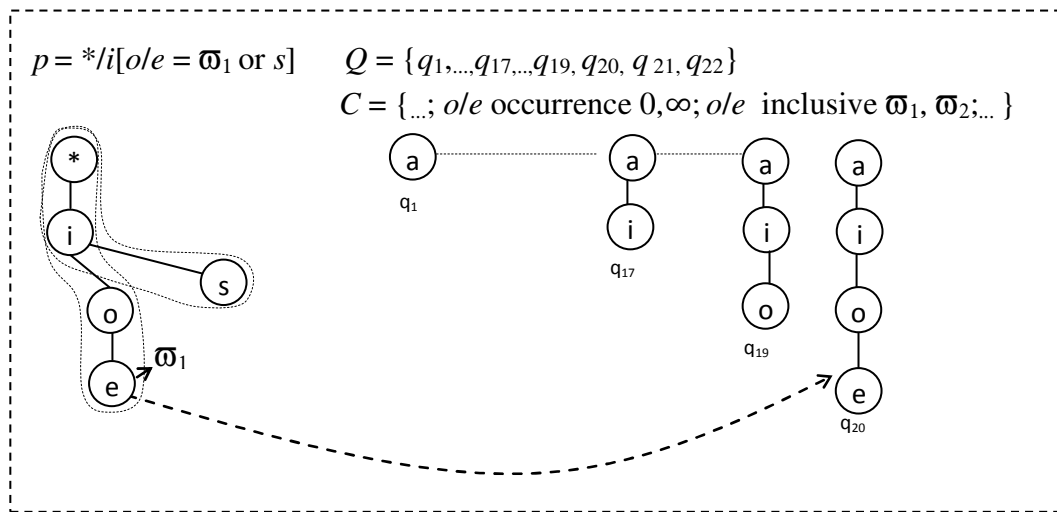


Figure 6.5 Sample XPath Path and Semantics for Semantic Predicate Reduction

The second condition s cannot be matched with any element in existing unique paths in Figure 6.5. Therefore the second condition is associated with a C_Q status. The existence of the second query condition allows the co-existence of the first query condition due to the connective OR. Hence the second condition is a mutual exclusive query condition to the whole XPath query.

Figure 6.6 summarises an XPath query, its components in terms of the query conditions and predicate as inputs for **conditionStatus** and output produced by the function.

XPath Query	Predicate Inputs	Outputs
p	$ePr = i[o/e = \varpi_1 \text{ or } s]$	$i[P_Q \text{ or } C_Q]$

Figure 6.6 Condition Status for Predicates

According to the proposed set of reduction semantic transformation rules **ST_{pe}**, rule **a** is used to remove the query condition s from p as the status σ satisfies a conflict-qualifier C_Q and connective Φ is OR, which allows it to be a mutual exclusive query condition in p . The semantic XPath query is produced as follow:

$$S = */i[o/e=\mathfrak{O}_1]$$

A running example based on the DBLP schema given in Appendix 1 is now used to demonstrate query conditions whose status are partial-qualifier and conflict-qualifier respectively.

Example. Select all PhD theses, which has been published in the year 1999 or have valid urls.

$$\text{XPath query } p = \text{dblp/phdthesis}[\text{year} = 1999 \text{ or urls}]$$

As shown in p , the predicate $Pr = [\text{year} = 1999 \text{ or urls}]$ has two query conditions **year = 1999** and **urls**. After being processed by function **conditionStatus**, a unique path entry **phdthesis/year inclusive 1950, 2020** is located in C list and **phdthesis/year** as part of unique path **dblp/phdthesis/year** located in Q list. Query condition **urls** cannot be matched to any element in unique paths in Q . Hence, function **conditionStatus** is associated with the first query condition with P_Q because the comparison value is an exact matched value within the restricted value range. The second query condition is associated with C_Q because it is a mutually exclusive query condition.

By following reduction semantic transformation rule **ST_{pr}**, rule **a** is used to remove the second query condition due to its conflict-qualifier status and the connective OR that makes the query condition a mutual exclusive query condition. The semantic XPath query is produced as below:

$$\text{dblp/phdthesis}[\text{year} = 1999]$$

Another running example is now considered to demonstrate the existence of query conditions whose status is full-qualifier and partial-qualifier respectively.

Example. Selects all PhD theses that have been published in 1999 or supervisor and title must be valid.

XPath query p where

$$p = \text{dblp/phdthesis}[\text{year} = 1999 \text{ or supervisor} = \text{dblp/phdthesis}/@id \text{ and title}/tn]$$

As shown in p , predicate $Pr = [\text{year} = 1999 \text{ or supervisor} = \text{dblp/phdthesis}/@id \text{ and title}/tn]$ has three query conditions including **year=1999**, **supervisor =dblp/phdthesis/@id** and **title/tn**. After being processed by the **conditionStatus**, **phdthesis/year inclusive 1950 2020** is located in Q , and **phdthesis/supervisor = /dblp/phdthesis/@id** in **phdthesis/title/tn occurrence 1 1** is located in C . As a result, the first query condition is associated with a partial-qualifier as the comparison value **1999** is an exact value within the range 1950 and 2020 of **year**. The second query condition is associated with a full-qualifier status as the comparison value of the reference path is an exact match value for the **supervisor** defined in the schema. The third query condition is associated with a full-qualifier status as each Ph.D. A thesis must have at least one title as defined in the schema.

The required parameters and their values are now extracted as follows

$$ePr = \text{phdthesis}[\text{year} = 1999 \text{ or supervisor} = \text{dblp/phdthesis}/@key \text{ and title}/tn],$$

$\theta_2 = \text{year} = 1999, \theta_2 = \text{supervisor} = \text{dblp/phdthesis}/@key, \theta_3 = \text{title}/tn$ whose status are P_Q, F_Q and F_Q respectively.

Rule **c** in semantic transformation \mathbf{ST}_{pr} is used to determined the second and third query conditions are associated with F_Q and connective is AND. In addition, the comparison elements in both query conditions have minimal occurrence greater than 0, and the transformation rule produces semantic XPath query **dblp/phdthesis[year=1999]**.

6.3.4 Proposed Functions

This section proposes a set of functions including **transformPredicate**, **conditionStatus**, **nPCV**, **yPCV** and **getOccurrenceValue**. These functions

consolidate all the rules proposed in 6.3.1, 6.3.2 and 6.2.3. Before each function above is described in details, their important roles are summarized below.

The function **transformPredicate** is called by the main algorithm when predicates are detected in the XPath query. The function **transformPredicate** first calls the **conditionStatus** function to obtain the status of each query condition in the predicate. The **nPCV** and **yPCV** functions are called through the **conditionStatus** function according to the type of the comparison values of each condition. Function **nPCV** handles the checking of a condition whose comparison value is not a path value. Instead the condition may be a constant for example **article[1]** where 1 is the constant, or no value, **phdthesis[title]** where **title** is the condition with no value. Function **yPCV** handles the checking of the condition whose comparison value is a path value.

The **getOccurrenceValue** function is used to obtain the minimal occurrence value of the branching element so that it can determine whether the whole predicate can be removed.

Function 1: **transformPredicates** first calls the function **conditionStatus** to determine the status of query conditions in a predicate (Line 1:1), in which the results are stored in *resultCondition*. Once the *resultCondition* has been obtained, the semantic rules, proposed in sections 6.3.2 and 6.3.3, are implemented and applied (Lines 1:2-1:9).

The predicate is eliminated when the final *resultCondition* is empty because all the query conditions have been removed (Line 1:10). Otherwise the predicate is regarded as predicate reduction (Line 1:3-1:8) because only some query conditions have been removed. On the other hand, the predicate can be completely in conflict, that is when there is a query condition associated with C_Q status. For example, a query condition has status C_Q and is connected by AND to the leading query condition and following query conditions are given a status other than C_Q such as F_Q or P_Q (Lines 1:11-1:12).

Function 1:transformPredicates (String ePr, String p)

```

    Let resultCondition be an empty list,  $\phi = \{OR, AND\}$  be connective, Q be the list of unique path q,  $F_Q$  be full-qualifier status,  $P_Q$  be partial-qualifier status
1:1  resultCondition = conditionStatus(predLocation, p)
1:2  FOR each condition in resultCondition
1:3  IF (condition is ' $F_Q$ ' && no  $\phi$ ) || (conditions is ' $F_Q$ ' &&  $\phi$  is OR) THEN
1:4      Remove condition from resultCondition
1:5  ELSEIF (condition is ' $F_Q$ ' &&  $\phi$ ) || (condition and neighbour project different comparison element) THEN
1:6      Remove condition from ResultCondition
1:7  ELSEIF (condition is ' $C_Q$ ' &&  $\phi$  is OR) THEN
1:8      Remove condition from ResultCondition
1:9  END LOOP
1:10 IF resultCondition is NULL THEN remove predicate [ ] from ePr
1:11 ELSEIF resultCondition is not NULL THEN check IF there is  $C_Q$  which is join by AND
1:12     among conditions with ' $F_Q$ ' or ' $P_Q$ ' THEN ePr = NULL
1:13 RETURN ePr

```

The *resultCondition* is updated with a set of statuses such as F_Q , P_Q or C_Q together with OR or AND if any exists. The transformation will then decide if the predicate is eliminated completely or modified.

Function 2: **conditionStatus** performs tasks to determine the statuses of query conditions by first accepting the input *ePr* and XPath query *p*. It then evaluates query condition(s) to determine the status of each query condition by using information of *Q* and *C*. *Pr* is a predicate that holds a set of query conditions ω such that $[\omega_1 (o) \omega_2 (o) \dots (o) \omega_n]$, and *o* is the connective represented for AND/OR; *e* is a branching element of a predicate.

Each query condition in the predicate is evaluated based on several principles. The comparison operator τ is detected to determine the existence of a comparison value in the query condition. The query condition is then verified against the unique paths in list *Q*. The *resultCondition* is then updated with ' F_Q ' for the current query condition if no τ has been detected and the comparison element or path fragment is found in *Q* (Lines 2:3 – 2:4). When AND or OR is detected after each processed query condition it is brought over to *resultCondition* (Line 2:2).

Function 2: conditionStatus (String ePr, String p)

Let τ be $=, \leq, \geq, < \text{ or } >$, temp be a temp variable, e be outer element extract from ePr, condList be an empty list, ϕ be connectives {OR, AND}, ω be a condition in ePr, F_Q be full-qualifier status, P_Q be partial-qualifier status, C_Q be conflict-qualifier

```

2:1 FOR each  $\omega$  waiting to be evaluated in ePr
2:2 IF  $\phi$  is 'OR' or 'AND' update condList with  $\omega$ 
2:3 ELSEIF  $\omega$  is (a path expression or element) with no  $\tau$  THEN
2:4   Check for  $\omega$  in q && update resultCondition with ' $F_Q$ '
2:5 ELSEIF  $\omega$  is a value without element THEN check e exist in q of Q update resultCondition with ' $P_Q$ '
2:6 ELSEIF  $\tau$  exists in  $\omega$  THEN
2:7   extract the compareFragment and compareValue
2:8   IF compareValue contains no '/' or '/' THEN resultCondition = nPCV(e, compareElement,
        compareValue,  $\tau$ , resultCondition)
2:9   ELSEIF compareValue contains '/' or '/' THEN resultCondition = yPCV(e, compareElement,
        ePr, compareValue,  $\tau$ , resultCondition)
2:10 END LOOP
2:11 Find all those that have value and project same element && exists OR only
2:12 IF values match complete set of values for same element in C THEN
2:13   IF minimal occurrence of the element by calling function 4 is greater 1 THEN
2:14     Set these condition in resultCondition to ' $F_Q$ '
2:15 RETURN resultCondition

```

A query condition can also be a *context position index* value. A position index value normally refers to the context element for a certain index value, for example, as in the XPath query `//article[1]`.

The predicate `[1]` placed on element **article** is, also known as a branching element. **1** is the context position index where the context element is **article** and the predicate is used selecting the first **article** only. When the predicate is specified with a query condition that has a context index value only, the context index value is based on the branching element. The transformation checks for the validity of the branching element in one of the q in Q . When element **article** in a valid q , that satisfies the structure of p is located, its status is set to P_Q (Line 2:5).

When a query condition is detected with a comparison operator τ , this means that a query condition is expected with a comparison path fragment or element and comparison value. Here we need to determine whether the comparison value is a path ('/' or '/') or a value such as a name or an integer. If comparison value is not a path then the transformation calls function **nPCV**, which is an abbreviation of a non-parent constraint value (Line 2:8). If the comparison value is a path value where '/'

or '/' detected in value, then the transformation calls function **nPCV**, an abbreviation for pattern constraint value (Line 2:9).

The transformation also takes care of the case where a set of query conditions carries status P_Q but they all project the same comparison element and their values match the complete set of values for the same element specified in C . These query conditions are set with F_Q (Lines 2:10-2:14). This case most likely applies to an enumeration constraint applied on an element.

Function **nPCV** accepts parameters such as comparison element $compare_e$, comparison values $compare_v$, comparison operator τ , and the *resultCondition*.

Function 3: List nPCV(String outerE, String compare_e, string compare_v, string τ , List condList)

```

    Let  $\tau$  be =,  $\leq$ ,  $\geq$ ,  $<$  or  $>$ ,  $F_Q$  be full-qualifier status,  $P_Q$  be partial-qualifier status,  $C_Q$  be conflict-qualifier,
    minO and maxO be occurrences, C be list of constraints of elements
3:1  FOR each c in C
3:2    Locate c contains comparee
3:3    IF (( $\tau$  is '=' && comparev match a value of set in c) ) THEN update condList with  $P_Q$ 
3:4    ELSEIF( $\tau$  not '=' && comparev satisfy full value set in c) THEN
3:5      IF (minO = getOccurrenceValue(outerE, comparee))>=1 THEN update condList with  $F_Q$ 
3:6      ELSEIF (minO = getOccurrenceValue(outerE, comparee))<1 THEN update condList with  $P_Q$ 
3:7    ELSE update condList with  $C_Q$ 
3:8  END LOOP
3:9  RETURN condList

```

The function first locates the constraint and values of comparison elements in list C (Line 3:2). Based on τ , the comparison value is determined whether it is an exact match value or within the match range which is set for comparison in list C . If τ is an equality operator and the comparison value matches only part of the value set in list C then query condition is associated with a status P_Q (Line 3:3). Otherwise if τ indicates with an operator other than an equality or non-equality operator, that means the values of comparison element cover a restricted range value set in list C and then the comparison element needs to confirm its minimal occurrence before F_Q is associated with it. If the minimal occurrence value of the comparison element is greater than 0, this indicates the existence of data in the databases for the comparison element. The query condition can be associated with F_Q . Otherwise the query condition is associated with P_Q (Lines 3:4-3:6). The query condition is set to C_Q if a comparison element or values are not satisfied as specified in the list C (Line 3:7);

where “not satisfied” means the comparison element cannot be located or values do not match.

Note that a comparison element may have more than one constraint. For example an element may be restricted by an enumeration constraint and occurrence constraint in which the occurrence constraint is compulsory.

Function **yPCV** accepts parameters such as a comparison path fragment, comparison value, τ , and the *resultCondition*. The function first locates the constraint and values in list *C* based on the comparison path fragment (Line 4:2). When comparison operator τ is an equality operator and if the comparison path value expected to match the path value of the path fragment in list *C*, then the query condition status is set to F_Q if the minimal occurrence of comparison path is greater than 0. Otherwise the query condition needs to be set to P_Q if the minimal occurrence is less than 1 (Lines 4:3-4:5). If the comparison path fragment and the value cannot be found then the query condition is set to C_Q (Line 4:6). The *condiList* is returned to the function that calls **Function 4**.

Function 4: List yPCV(String outterE, String compare_e, string compare_v, string τ , List condiList)

Let C_Q be conflict-qualifier, F_Q be full-qualifier, P_Q be partial-qualifier, minO and maxO be occurrences,
C be list of constraints of elements

```

4:1 FOR each c in C
4:2   Locate c contains comparee
4:3   IF ( $\tau$  is '=' && c satisfy comparev) THEN
4:4     IF (minO = getOccurrenceValue(outterE, comparee))>=1 THEN update condiList with  $F_Q$ 
4:5     ELSEIF (minO = getOccurrenceValue(outterE, comparee))<1 THEN update condiList with  $P_Q$ 
4:6   ELSE update condiList with  $C_Q$ 
4:7 END LOOP
4:8 RETURN condiList

```

Function **getOccurrenceValue** performs a simple task to retrieve the minimal occurrence values for a comparison element or a path fragment by locating them in list *C* and returns them to calling function (Lines 5:1-5:5).

Function 5:getOccurrenceValue(String outterE, String compare_e, integer oValue)

```

5:1 FOR each c in C
5:2   Locate c contains outterE/comparee
5:3   Obtain the lower bound value of occurrence of outterE/comparee THEN exist
5:4 END LOOP
5:5 RETURN oValue

```

6.4 Summary

In this chapter, we have proposed predicate elimination and reduction semantic transformations. A predicate in an XPath query expresses a query condition to be fulfilled in addition to the structural constraint imposed by the path itself. The query condition is a Boolean expression. It may involve comparisons between elements and values, path expressions denoting elements to be compared as well as further path expressions. Due to a variety of predicates which have been given in a query, this chapter has proposed techniques to first determine the status of the query condition such as full-qualifier, partial-qualifier or conflict-qualifier. Once the query conditions are associated with appropriate status, semantic transformation rules for predicate elimination or predicate reduction are applied to transform the XPath query.

Each semantic transformation produces

- a semantic XPath query with no predicate. This is when the predicate is completely removed; or
- a semantic XPath query with a presence of a predicate. That is when a predicate is reduced by removing some, but not all, query conditions; or
- a semantic XPath query cannot be produced due to the conflict detected in the whole predicate; that is either every query condition produces conflict or some query conditions produce a conflict. Query conditions have connectivity AND may cause the whole predicate conflict if one of them is associated with a conflict-qualifier. If the conflict makes the semantic path NULL, then the transformation of the whole XPath query is terminated in the main program. Semantic transformation is the best way to avoid a conflicting XPath query being sent to the database, thereby wasting resources unnecessarily.

Chapter 7

Experimental Design

The aim of this chapter is to present an experimental design method that formulates the elementary experiment for conducting an evaluation to study the impact of query performance of XPath queries before and after they are transformed.

Two experiments are designed to evaluate the proposed semantic transformations presented in Chapters 4, 5 and 6.

The first experiment measures the query performance of a set of customized XPath queries and their semantic counter-parts using the DBLP data set. The second experimentation measures a set of micro-benchmarks (also known as the Michigan benchmark [Runapongsa et al., 2006]) XPath queries and their semantic counter-parts using the Michigan data set.

7.1 Experiment Design: A Background

Chapters 4, 5 and 6 proposed a series of semantic transformations using the available semantics in given XML Schemas to transform XPath queries to equivalent semantic XPath queries for optimization purposes. The proposed semantic transformations are intended to address the potential of semantics defined in XML Schemas (XSD) to overcome the limitations of semantics defined in Documentation Type Definition (DTD) used in semantic XML query optimization prior to this work.

7.1.1 Experiment Objectives

There are three main contributions of semantic transformations including semantic path transformations, semantic transformations for XPath queries specified with axes, and semantic transformations for XPath queries specified with predicates. For each main contribution, a set of semantic transformation rules have been proposed. Each rule is automatically determined and applied to a given XPath query. The semantic transformation rules are platform-independent. This means that they are capable of transforming XPath queries without needing to rely on the database platform. Details of the experiment strategies and operations are discussed in the following sections.

7.1.2 Experiment Evaluation Objectives

To achieve the experiment objectives, the experimental design is conducted using the following objectives:

- a. To identify the implementation of semantic transformation framework. That is, the implementation can be used for evaluation purposes, under the scope of query transformation for optimization purposes, within the availability of hardware and software.
- b. To choose a database platform (either native XML data storage or XML-Enabled data storage), where research must determine the availability of minimal requirements such as the XML Schema validation feature that is provided by the database platform.
- c. To select appropriate datasets and queries, the research must review several available real data sets to compare and investigate their completeness and limitations in terms of semantics, data structures and query processing coverage. This research also studies existing micro-benchmark data and queries to address the gaps and limitations that real data sets cannot comprehend.
- d. To define and unify variable names such as metrics, measuring units and computation procedures for two experimentations; this is to enable a logical

query performance comparison to be achieved across a range of relevant variables in the same system and environment.

- e. To describe the operational environment, in which experiments take place, including hardware configuration, software specification, system modules and the system capability such as the front-end and back-end interface (e.g. support connectivity to the database platform).
- f. To obtain and analyse the results of query performance. The conclusion must be drawn for the performance and limitation so that they can be used to identify the potential of semantic transformations as semantic query optimization devices.

Several of the set-ups are common to both experiments; however, others need to be designed according to the particular requirements of each experiment. In the next section, the set-ups common to the two experiments are described.

7.2 Common Setup for Experiments

The two experiments are concerned with evaluating the semantic transformation rules when applied to given sets of XPath queries. Section 7.2 details the several set-ups and decision making processes common to the two experiments.

7.2.1 Implementation Framework & Platform

The implementation of semantic transformation rules (or proposed algorithms) can be evaluated as an independent application module from the database platform. This means that the transformation can work without needing to know in advance whether the type of the database platform is either an XML-native or non-native database. Due to the current predominance of relational databases, most of the major relational database systems are available with an XML-enabled feature. For our experiment, a decision was made to select one of the leading commercial relational database systems.

7.2.2 Supporting of Minimal Requirement

Because of the decision to select a relational database system with an XML-enabled feature, a minimal requirement for storing experiment data must be met. That is, regardless of the technique used to store the experimental XML data, the XML Schema must be able to reside on the database so that the data can be checked for consistencies and conformation.

7.2.3 Choice of Experiment Data and Schema

Semantic query transformations require semantics defined in the schemas in order to transform the given queries to equivalent queries. We select XML data for experimentation based on various factors, but the most important one is semantics provided by its associated XML Schema. The more the semantics are available in the XML schemas, the more successful the evaluation will be.

When the data volume is significantly large and the semantics are insufficient, only certain semantic transformations can be applied. If the semantics are too rich but the volume of data is insubstantial, then the impact of the query performance study may not be effective.

Having considered the aforementioned factors of semantics and data sizes, we choose to adopt two sets of data. One is a real data set with an accompanying schema, namely DBLP [Ley, 2011]; the other is the micro-benchmark data set with an accompanying schema and a set of XPath queries [Runapongsa et al., 2006].

a. Real Data: DBLP

DBLP is the online resource providing bibliographies and subfield information on computer science books, conference proceedings, journals and so on [Ley, 2011]. DBLP data is selected for three important reasons. The first is that the actual real-world data such as DBLP data provides us with an insight into data that has user characteristics and expectations. The second is that the semantics in the schema are sufficient and able to support our proposed semantic transformations. The third is that the data size satisfies our current experiment environment in terms of hardware.

As XML data structure is represented by a tree-structured model, the XPath query expressiveness depends on the data hierarchy and selection of query patterns specified with or without predicates on multiple elements to match XML documents. If an XPath query is specified with predicates, some complex conditions in predicates enable a Twig pattern query [Runapongsa et al., 2006, Che et al., 2011]. A twig pattern query is the core operation in querying tree-structured data. Although the semantics in DBLP schema are relatively rich, the data hierarchy is not expressive as it supports only a six-level data hierarchy. We are not able to specify the extreme expressive structural and complex joins selection by conditions such as twig pattern queries by a using DBLP data set. It is a challenge to find such features in the real dataset. For this reason, we take the option of utilizing existing micro-benchmark data.

b. Micro-benchmark Data: Michigan

Several XML database management systems such as Tamino XML Database [Software AG, 2009], Oracle DB 11g [Oracle, 2010] and DB2 pureXML [IBM, 2009] to name just a few, have been developed on various database platforms. Technically, these databases are constructed differently from one another and their storage management models also differ. The way in which each individual system works and behaves has attracted an increasing interest from both industry and the research world. Hence, a great amount of effort has been put into developing XML benchmarking such as XMark [Schmidt et al., 2002], X007 benchmark [Li, et al., 2001], XBench [Yao and Ozsu, 2002], Michigan benchmark [Runapongsa, et al., 2006], TPoX [Nicola et al., 2007] and many more. Most of the existing benchmarking systems listed here are known as *application* benchmarks, except for the Michigan benchmark which is known as a *micro* benchmark.

The roles of application benchmarks focus on assessing the performance of given XML database systems by performing a large number of tasks such as selection, access method, modification, computation and insertion. These benchmarking systems provide an indication of performance to the database systems for potential users so that they can set their expectation for their applications.

On the other hand, there are various micro-benchmark systems that focus on generic database operations such as selection, computation and joins. A micro-benchmark system can provide an insight into generic database operations that assist the developers to understand and evaluate query operations which are significant or perform poorly.

We find that both data and XPath queries provided by the Michigan benchmark [Runapongsa et al., 2006] fulfill our requirements of evaluating expressive structural queries and complex selection query predicates. Because the Michigan data set provides a significant data hierarchy which is accompanied by an available schema, their XPath queries support much more complex joins than those that can be accomplished with the DBLP data set. Throughout the rest of the chapter, the Michigan benchmark is referred to as the micro-benchmark.

7.2.4 Setup of Operational Hardware, Software and System Modules

Figure 7.1 provides an overview of the system and implementation phases before semantic transformations take place. On the left of the vertical dash-line, the phase of schema registration is indicated by **2**, data validation is indicated by **1** and loading of XML data into the XML database is indicated by **3**. On the right of the vertical dash-line, the module for preprocessing semantics in XML Schema is indicated by **4**, the semantic transformations module is indicated by **5**, and the execution module of both the original and semantic XPath queries on the database is indicated by **6**. Task **7** returns an informative message to the user if the XPath query cannot be transformed. Task **8** returns a valid performance measurement to the users for evaluation. The arrows show the inter-relationships among the tasks.

a. Set-up of Hardware and Software

The two experiments are conducted in the same environment. The following hardware and software setup was used for both experiments:

- (1) The *hardware* includes a machine that has a configuration of Intel® Core™ Duo E7300 2.66 GHz. 2.67GHz and 2.0 GB of RAM; and

(2) the *software* includes Windows 7 Professional OS, Java VM 1.6 and Matlab 2009.

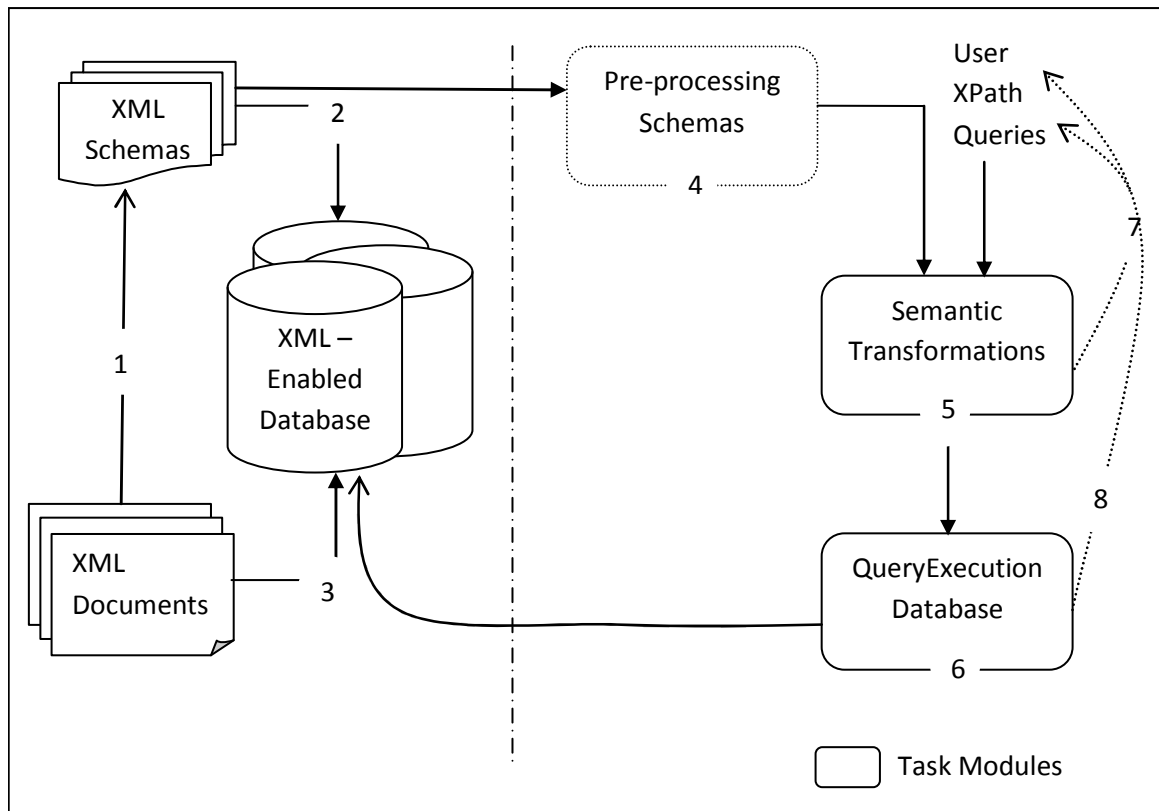


Figure 7.1 Overview of System Architecture and Task Modules

All semantic transformation algorithms are written and compiled in Java while all XPath queries are executed on a prominent off-the-shelf commercial relational database with an XML-enabled feature. Our semantic transformations can be run as add-on devices for any existing database as long as an XML capability with schema-awareness facility is supported. We adopt an out-of-box installation for our chosen database platform. We do not perform any database configuration such as indices or specific joins.

b. Setup of System Modules and Detailed Description

Upon the completion of XML Schema registration, data loading and verification, the semantic transformations can take place. The effectiveness of semantic transformations greatly depends on how the semantics are derived and managed. We have proposed our semantic derivation technique to derive and store semantics in

Chapter 4, Section 4.1. We derive and pre-process the semantics prior to beginning of the semantic transformation process. This initial pre-processing of semantics is time consuming; however, this is worthwhile because we need to accomplish this task only once and the derived semantics can be used for the transformation of all XPath queries until the semantic transformation application is terminated. By doing such a pre-processing of semantics we can avoid the overhead of searching and processing the semantics in the XML Schema during the transformation stage.

Preprocessing Schema Module: we now present an algorithm that implements the semantic derivation technique proposed in section 4.1 in Chapter 4.

Algorithm **PreprocessingSchema** accepts two input parameters T and R where R is the root name of the schema and T is the file name of the schema (Line 1). The main objective of the algorithm is to produce lists Q and C (Line 2). Q contains a list of unique paths and C contains a list of constraints (together with their associated values) of elements. Parameters such as *par*, *element*, *ePair* and *cName* are defined (Lines 4 - 5). Parameter *cName* is a list that contains predefined constraint names. The constraint names can be increased at any time. Currently, we store constraint names based on the DBLP schema.

At start-up, the algorithm will require the input parameters R and T . It then starts reading the schema file and checks if the root R is valid. If the root is valid, then the first item added to the C list is the root and occurrence constraint as well as its values (Line 8). It sets the flag *foundR* to *true* and sets the root as the parent and allows the processing to progress further (Line 9-10). If the given root is not found, the schema cannot be processed (Line 24).

For each line read from the schema, the process first searches for an element and its constraints. If the element is a complex type, then it sets the element as the parent (Line 12) and processes the constraints of the element (Line 18) by calling the **Function** `List constraintValues`.

The children of the complex type are processed (Lines 13-20) by building up the *ePair* list where each item is expressed as *parent/child* e.g. *dblp/article*. Within the building block of *parent/child*, the child element is also identified as either an

attribute element or non-attribute element so that the information can be prepared properly. For example, if it is an attribute, then it should be *parent/@attribute* (Line 19). If it is a non-attribute element, then its constraints such as *occurrence*, *inclusive*, *exclusive*, etc are processed to build up list *C* in which **Function** *List constraintValues* is called (Line 18). Once it reaches the end of the schemas, two lists, *ePair* and *C*, are built. Upon complete reading of the schema file, the algorithm produces *R* and *ePair*, which are then used to construct list *uniquePath* (Lines 21-23).

Algorithm 1: PreprocessingSchema (Schema S, String root)

```

1  Input : T = XSD schema; R = root name of the schema
2  Output: Q = List of sequence elements defined in XSD Schema; C = Semantic knowledge of elements obtained from T
3  Begin
4    Let par be parent, element be child, Let ePair be a list to contain pairs of elements that has a parent/child relationship
5    Let cName = {keyref, key, enumeration, inclusive, exclusive, pattern, sequence, choice, all, length, attribute, occurrence}
6    While not end of file (T)
7      read next line
8      If found R in / Then
9        foundR=true; push R + " Occurrence=(1,1)"
10     If (foundR) Then
11       Extract element from /
12       If element is a type Then par = element
13       Do
14         Read next line /
15         Extract element from /
16         If element is an attribute type Then element = @element
17         Push par/element to ePair
18         If / contains cname in cName Then C = constraintValues(/, C, List ePair, cname)
19       Until end of type
20   Loop
21   If (foundR) {
22     Push R into Q
23     Q = UniquePath(ePair, Q)
24   Else "invalid root. Nothing derived".
25   End
26   Function List UniquePath(List pair, List Q)
27     For each item x in pair
28       For each item y in pair
29         Push x+"@"+y into uP if found leftmost element in y occurs in x as right most element
30       End For
31     End For
32     Return Q
33   End Function
34   Function List constraintValues (String Line, List C, List ePair, String cName)
35     Let vals be a set of values of cName extracted from Line, e be element of cName
36     extend e to es using ePair until es is unique in c
37     Push es constrName=(vals) to C
38     Return C
39   End Function

```

The **Function** *List constraintValues* (Lines 34-39) accepts the input such as the line *line* that contains information about the element including name, constraint name and value of constraint. The ancestors of the element *e* needing to be

constrained are extended to further ancestors using an *ePair* list to make it unique in *C* (Line 36). The item *c* is then formed to carry the sequence of new extended elements including constraint element and name of constraints as well as the values of the constraint. New item *c* is placed into list *C* (Line 37).

The next step is to construct the unique path *Q* list. The root is first added to the *Q* list with root as the first unique path and then starts to process the remaining unique paths using items in the *ePair* list. The first item in the *ePair* should have the parent as root and child is the left most element on the next level as was proposed in chapter 4, section 4.1. For example, in the DBLP schema, *dblp* is the parent. The first item in the *ePair* list is *dblp/article*.

The **Function** `UniquePath` accepts the *ePair* and *R* values (Lines 26 – 32). This function is a guideline for simultaneously processing and sorting the unique path. It basically uses the proposed breath-first search direction as proposed in Chapter 4, Section 4.1 to derive all unique paths.

The pre-processing schema module performs the task only once at start-up and is terminated once it completes the processing of all input schemas, as indicated by module number 4 enclosed by the dotted line in Figure 7.1. The Semantic Transformation module then takes control as indicated by module number 5 in Figure 7.1. If any changes are made to the XML Schemas, this module will need to restart (though we do not expect the structure of the schema to change frequently).

Semantic Transformation Module is the one that implements all the algorithms proposed in Chapters 4, 5 and 6; this is a real-time module. It is called after the pre-processing schema module has successfully produced a unique path list *Q* and constraints of elements list *C*. It continually detects the users' XPath queries and transforms them. This module will return valid semantic XPath queries and also send the queries to the database for execution; hence, it is called the Query Execution Database module. The Semantic Transformation module logs the transformation time and valid semantic XPath query in a dynamic file so that the user can manipulate this information independently. We fetch this information to plot results in MatLab.

If any conflicts of constraints are detected in an XPath query, a message is returned to inform the user as indicated by an upward arrow labeled with number **7** in Figure 7.1. The returned message is logged in a dynamic text file and will be used to measure the performance of those queries that return no result.

For a valid transformed XPath query, the transformation time is tracked so that it can be added to the average execution time of the semantic XPath query to calculate the *total performance time*. The average execution time is the total execution time of an XPath query divided by the number of executions. The performance of the semantic transformation XPath query is measured by the total performance time.

Query Execution Database Module sends an XPath query and its semantic XPath query to access the required information in the database, as indicated by module **6** in Figure 7.1. Once the required information is retrieved, it is returned directly to the user instead of via the XML transformation module as indicated by task module **8** in Figure 7.1. This module can be called by the Semantic Transformation module or it can work independently.

As the Semantic Transformation module also provides semantic XPath queries and their transformation time in a dynamic text file, the query execution database module can be executed independently using information in the dynamic file.

7.2.5 Computation Procedure and Metrics

Computation Procedure. Each original/semantic XPath query is executed for five runs. The average execution time is calculated based on the last 3 runs. This is achieved in module **6** in Figure 7.1. By ignoring the first two runs and calculating the average execution time of the last three runs, this supports both cold and hot warm-ups that prevents any data buffering problem from previous runs.

Metrics. For each pair of queries (original XPath query and its semantic XPath query), the experiment compares the total performance time (in milliseconds).

For the original XPath query, the execution time of query performance is measured by the average execution time as calculated by the computation procedure.

As explained earlier in the Semantic Transformation module, for the semantic XPath query, the execution time of query performance is measured by the average execution time plus the transformation time as calculated by the computation procedure.

7.3 Individual Setup for Individual Experiment

This section describes the individual experiment set-up including semantic enhancement, data scaling, data cleansing, query design, metrics and computation procedures.

7.3.1 Experiment 1: Using DBLP Data

This section describes the process of enhancing semantics in XML Schema that enables data cleansing, data scaling sets and query design. Metrics and computation procedures are also described here for individual experimentation.

a. Semantics Enhancement for XML Schema

The actual DBLP schema is in DTD format which we have decided to convert to DBLP.xsd. This is because our semantic transformations support more semantics than what DBLP.dtd can offer. To enable all semantic transformations to work with this DBLP schema, all semantics offered in the DTD Schema remain. In addition, the semantics can be enriched by adding further semantics so that we can demonstrate the whole range of our semantic transformations. The alteration of the DTD Schema must be kept as simple as possible so that the verification of data will take less effort. One advantage of adopting the DBLP is that we can also prove that our semantic transformations are not limited to DTD. The proposed semantic transformations enable us to take care of semantics in both DTD and XML Schemas (xsd).

In summary, the depth of the DBLP XML Schema is 5 and the second hierarchy has a breadth of 7 schema elements. Each different labeled schema element on the second hierarchy must have at least one data node. The rest of the hierarchies depend on the occurrence constraint of each labeled element. We convert from DBLP.dtd to

DBLP.xsd and maintain all the existing semantics that have been transferred from the DBLP.dtd. At the same time, we also add a number of constraints to achieve our semantic transformations. The semantics are modified and added as follows:

First, the data type of year element is changed from string to integer. Then the year is restricted with a range of values between 1950 and 2020. This range of values is not compulsory; however, it is good to have to demonstrate the usefulness and practicality of predicate elimination semantic transformation. In this schema, year element is a global element whose setting is applied throughout the schema. For each PhD thesis, there must be at least one supervisor who is identified by his/her PhD qualifier. To achieve this, we use a key reference for the supervisor. That is, a supervisor is referenced by an existing thesis but not by the current thesis. This is to prevent the supervisor from being confused with the student undertaking the PhD thesis.

For each element on the second hierarchy in the schema, we modify the order constraint by replacing the ‘choice’ with a ‘sequence’ value for ‘dblp’, ‘article’, ‘inproceedings’, ‘proceedings’, ‘book’, ‘incollection’, ‘phdthesis’, ‘www’. This is to ensure data such as ‘article’, ‘inproceedings’, ‘proceedings’, ‘book’, ‘incollection’, ‘phdthesis’, ‘www’, ‘author’, ‘title’, ‘pages’, ‘year’,..., ‘url’ are in the order as set out in the schema. For the occurrence constraint, the minimal occurrence is set to for elements including ‘dblp’, ‘article’, ‘inproceedings’, ‘proceedings’, ‘book’, ‘incollection’, ‘phdthesis’, ‘www’, ‘author’, ‘title’, title name, and ‘year’. The occurrence for these elements varies between 1 and infinite.

XML data in Figure 7.2 conforms to the DBLP XML Schema. The schema is rich in semantics of elements which enables semantic transformations to be applied to a wider range of XPath queries.

b. Data Cleansing

Once the XML Schema has been changed, we also need to change the data in order to achieve data conformation. Data cleansing is important for DBLP data sets as we have made alterations to the semantics in the DBLP XML Schema. Therefore, we have developed a data cleansing module (written in Java) to eliminate the

inconsistencies of data according to the modified semantics. For example, the new data order based on the *sequence* property of the *order* constraint is enforced on elements in the order of **article**, **inproceedings**, **proceedings**, **book**, **incollection**, **phdthesis**, **www** from left to right and they are the children of **dblp**. The *order* constraint is also applicable to children of some of these elements.

In addition to the *order* constraint, the data of element **supervisor** also needed to be checked for correct values that are set in the XML Schema. The values of **year** are also needed to be ensured within the specified range. The data cleansing module also validates the occurrence data for all the elements according to the changes that have been made in the XML Schema.

To ensure the conformation of the new data after cleansing, a validation tool such as XMLSpy is used to validate the new data against the XML DBLP Schema.

Data scaling is important. Good experiment data with regards to a performance study must be able to scale to several data sets if we are to examine the query components that respond to various data sizes. There are several scaling options that one can follow to tailor the data sets. For our experiment, we need several incremental data sets.

c. Data Scaling

We choose two DBLP data sets of 350 and 100 mega bytes (MB). The first data set 350MB is referred to as Experiment Data Group 1 and the second data set of 100MB of Experiment Data Group 2. We find that XPath queries specified with XPath axes are unstable or cannot be executed most of the time when the data set of 350MB is tested. We suspect this is due to a combined limitation of XPath axes supported by our chosen DBMS with XML-enabled feature and availability of the hardware. For this reason, we have to reduce the size of the data set (Experiment Data Group 2) until the XPath queries specified with XPath axes show a stable processing.

Each data set (DS) is scaled down to 10 incremental data sets: 0.1DS, 0.2DS, 0.3DS, 0.4DS, 0.5DS, 0.6DS, 0.7DS, 0.8DS, 0.9DS, 1DS. Hence, Experiment Data Group 1 and Experiment Data Group 2 will each have 10 scaled data sets.

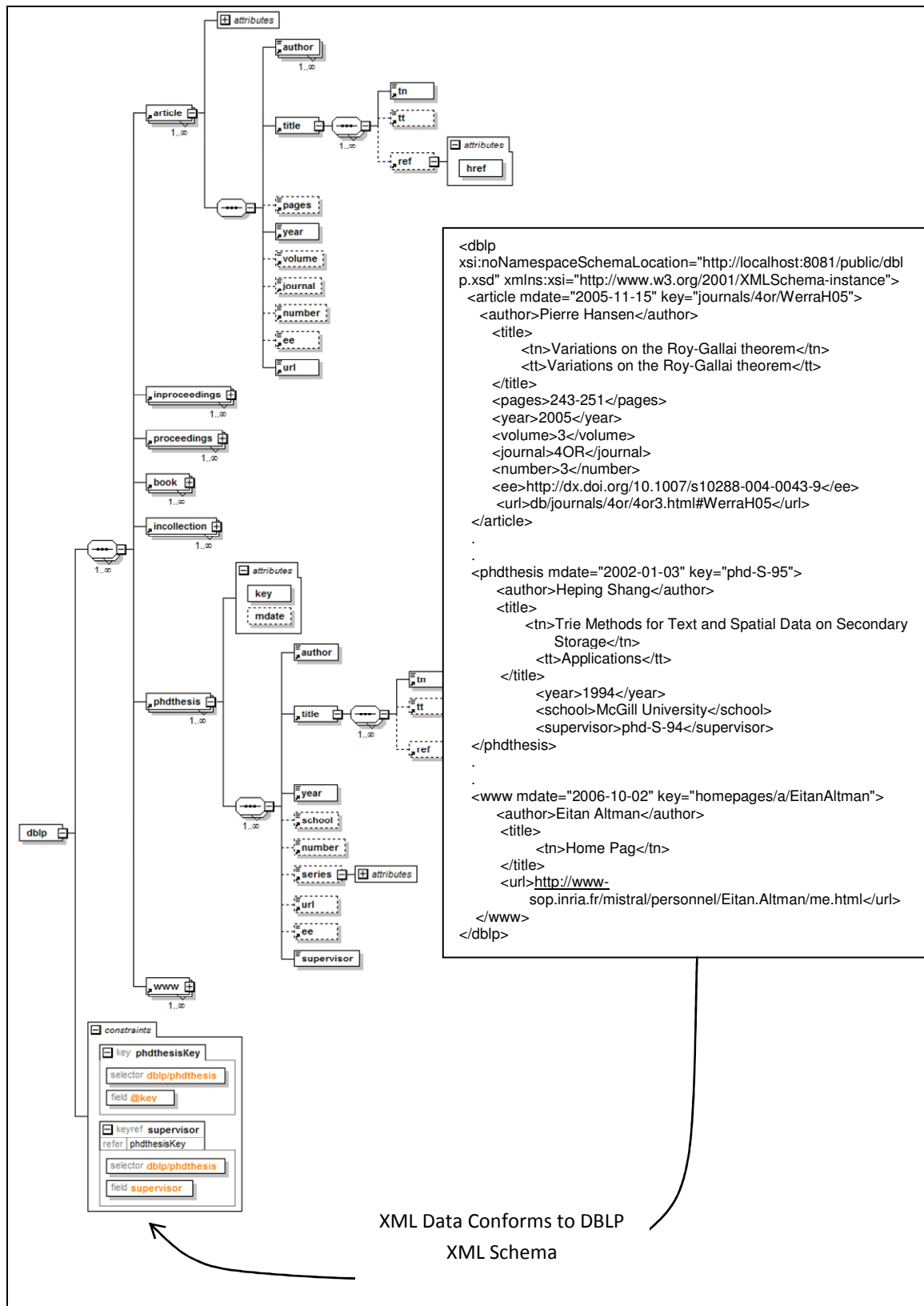


Figure 7.2 DBLP XML Schema and Snapshot of XML Data (after modification)

As we do not change the depth and the breadth of the schema, each data set is scaled by scaling the number of data nodes on the second hierarchies. There are 7 schema elements on the second hierarchy including **article**, **inproceeding**, **proceeding**, **incollection**, **book**, **phdthesis**, **www**. For each of these 7 element groups, we split the data into 10 sets.

Assume the element data group of **article** has 70MB, **inproceedings** has 50MB in the whole DBLP XML document of 350MB, then

article: (0.1DS) = 7MB, (0.2DS) = 14MB,..., (1DS) = 70MB

inproceedings: (0.1DS)=5MB, (0.2DS)=14MB,..., (1DS)=50MB.

Once we have 10 scaled sets for each element data group, we then compile the actual 10 DBLP incremental data sets by merging the seven element data sets in the order of schema elements. Hence, the first DBLP scaled data set (0.1DS) is the result of merging of **article**(0.1DS), **inproceedings**(0.1DS), **inproceedings**(0.1DS), ..., **www**(0.1DS), and then the second set of data is the result of merging of those with (0.2DS) in the order of schema elements and so forth.

We apply the same concept of scaling to the Experiment Data Group 2 sets. We develop a module, using Java, to automate this scaling process. For the actual data sizes and scales for each Experiment Data Group, refer to Figure 7.3.

Experiment Data Group 1			Experiment Data Group 2		
Data Set	% Scale on Data Nodes	Data Size (MB)	Data Set	% Scale on Data Nodes	Data Size(MB)
1	0.1	35	1	0.1	10
2	0.2	70	2	0.2	20
3	0.3	105	3	0.3	30
4	0.4	140	4	0.4	40
5	0.5	175	5	0.5	50
6	0.6	210	6	0.6	60
7	0.7	245	7	0.7	70
8	0.8	280	8	0.8	80
9	0.9	315	9	0.9	90
10	1	350	10	1	100

Figure 7.3 Scaled Data Sets for DBLP Data

10 sets of incremental size of DBLP data for experiment data groups 1 and 2, as shown in Figure 7.3, are loaded into a database platform with XML Schema validation enabled.

d. XPath Query Taxonomy

This section first provides the XPath query taxonomy; second, it recommends XPath query patterns that are suitable for each semantic transformation typology. The XPath query patterns suggested for the experiment are based on the XPath taxonomy. To simplify the subject matter, the XPath query taxonomy is either a **full-path** pattern or **partial-path** pattern (refer to chapter 3 for definitions of these patterns).

An XPath query has a full-path pattern if and only if it satisfies one of one or more of the following properties:

- XPath query is specified without any predicate []
- Path element is represented by a valid element name or wildcard ‘*’

Figure 7.4 shows four different types of full-path patterns; hence, no predicate exists in any of the four different XPath queries.

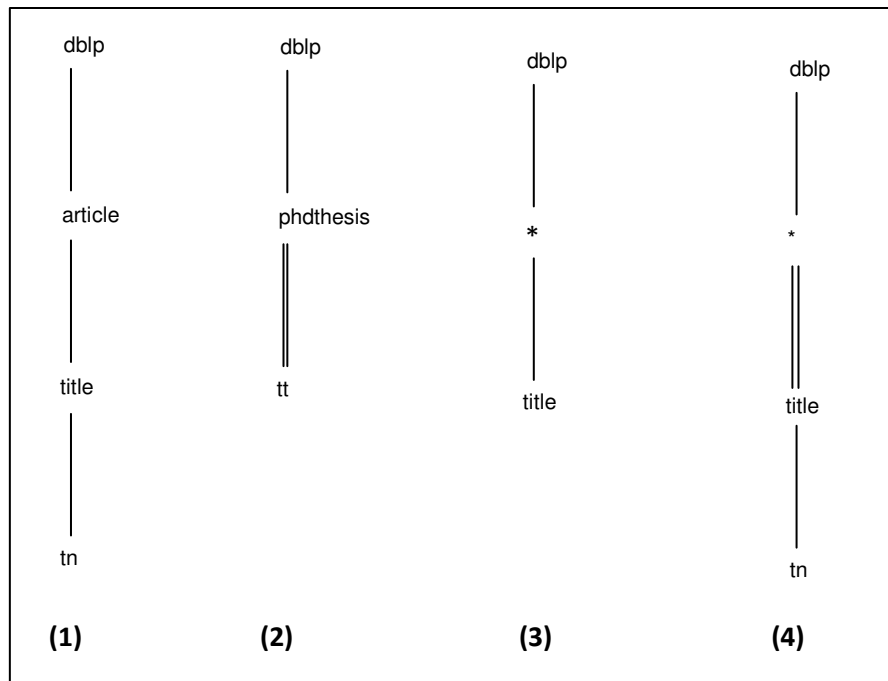


Figure 7.4 Full-path XPath Queries

In XPath query (1) **dblp**, **article**, **title** and **tn** have parent-child relationships. In XPath query (2) **dblp** and **phdthesis** have a parent-child relationship and **phdthesis** and **tt** have an ancestor-descendant relationship. In XPath query (3) **dblp**, * (wildcard), and **title** have only parent-child relationships. In XPath query (4) while parent-child relationships are between **dblp** and * (wildcard) and between **title** and **tn**, ancestor-descendant relationship exists between * (wildcard) and **title**.

An XPath query has a partial-path pattern if and only if it has a combination of the following and satisfies one or both of the following properties:

- Composition of full-path pattern properties
- Essentially defined with a composition of a predicate []

As mentioned earlier, predicates that exist in an XPath query indicates a twig pattern in the query. That is, a path pattern has a branching element, which has a further one or more children. We categorize a twig query as either a *simple twig* pattern XPath query or a *complex twig* pattern XPath query.

An XPath query has a simple twig pattern when a branching element has more than one child and each child has no further children.

An XPath query has a complex twig pattern when a branching element has more than one child and each child has further children or descendants.

Figure 7.5 show various types of partial-path XPath queries. In XPath query (5) **dblp**, **article**, **title** and **author** have parent-child relationships. The branching element **article** has two immediate children, **title** and **author**; none of them has further children. In XPath query (6) the branching element **dblp** has two immediate children, **article** and **author**, and none has further children. In XPath query (5), * is a branching element that has an immediate child, **year**, and descendant **tn**. In XPath, query (6) has more than one child and each child does not have any further children, making both (5) and (6) simple twig pattern queries.

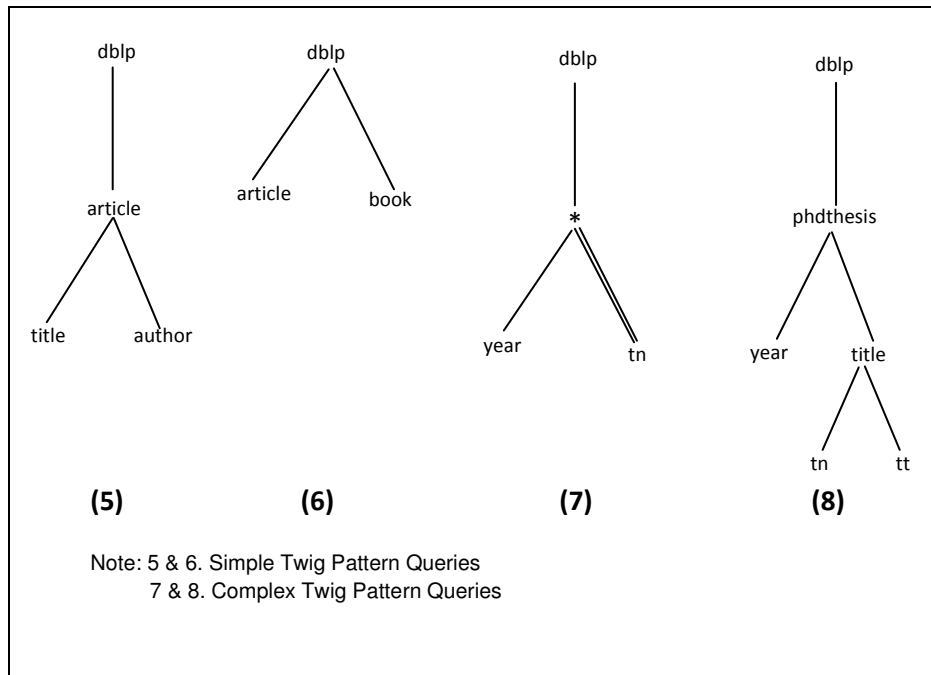


Figure 7.5 Partial-Path XPath Queries

In the XPath query tree (7), the branching element ***** has an immediate child **year** and a descendant **tn**. There is expected to be one or more hierarchical elements between wildcard ***** and element **tn**. Hence **tn** is not an immediate child of *****, therefore XPath query (7) is a complex twig pattern query. In an XPath query (8), the first branching element **phdthesis** has two immediate children **year** and **title**. The immediate child **title** has further children **tn** and **tt**, which makes XPath query (8) a complex twig pattern query.

For each semantic transformation category, we need to propose a set of XPath queries with regard to the query taxonomy above. By manipulating DBLP data sets and XML schema, we make it possible to issue queries with better XPath query patterns. XPath queries enable us accommodate a variety of XPath query components, which respond to our semantic transformations. Below, we provide the guidelines for XPath query patterns for each category of semantic transformations.

- **Semantic Path Transformations**

Based on the query taxonomy, the XPath queries suggested for semantic path transformation are as follows:

- *Semantic Path Expansion*: XPath query patterns are full-path patterns in which each XPath query is expressed with an ancestor-descendant and/or parent-child hierarchy and/or with the *wildcard* element ***.
- *Semantic Path Contraction*: XPath query patterns are full-path patterns in which each XPath query is expressed with an ancestor-descendant and/or parent-child hierarchy *//* and/or *wildcard* element ***.
- *Semantic Path Complement*: XPath query pattern is a simple twig query pattern that is implemented by the parent axis (or optional operator *..*).

- **Semantic Transformations for Axes**

XPath queries suggested for semantic transformations for XPath query axes are *full-path* patterns in which element(s) is specified with an XPath axis from {*child*, *self*, *parent*, *descendant*, *descendant-or-self*, *following*, *preceding*, *following-sibling*, *preceding-sibling*, *ancestor*, *ancestor-or-self*}.

- **Semantic Transformations for Predicates**

XPath query predicates support query conditions. Query conditions in a predicate joined by operators AND or OR are classified as value-based type or pointer-selected type [Runapongsa et al., 2006].

A value-based type condition compares the values of different elements and a value of an element that is either an attribute or a leaf node. A pointer-selected type compares the values of a path fragment and most likely returns a sub-tree instead of a single value. XPath queries specified with predicates for semantic transformations are partial-path patterns.

Semantic Predicate Elimination is applied to query conditions, which are connected by AND and/or OR operator(s). *Semantic Predicate Reduction* is applied to query conditions which are connected by an OR operator.

For the DBLP data sets, due to the shallowness of data hierarchies, complex twig pattern queries are very primitive. For this reason, more complex twig patterns are carried out in the second experiment.

7.3.2 Experiment 2: Using Benchmark Data

This section describes the semantics required for the experimentation, scaled data sets, query selection, metrics and the computation procedure.

a. Semantics in XML Schema

The micro-benchmark schema is available in both XSD and DTD schema formats. The micro-benchmark data set is adopted mainly because of its significant number of hierarchies of 16, which enables this experiment to evaluate the expressiveness of path expressions and complex predicates for selection of query type. There is no alteration made to the micro-benchmark XML Schema.

The micro-benchmark schema and its sample XML data are shown in Figure 7.6. The schema is constructed around a **BaseType** element which contains a set of attributes such as **aUnique1**, **aUnique2**, **aFour**, **aSixty**, **aSixteen**, **aLevel**, **aString**. The hierarchy is indicated by a level value of attribute **aLevel**. Each **Basedtype** element contains two setf of subelements such as **BaseType** and **OccasionalType**. The presence of the **OccasionalType** element is determined by the value 0 of the attribute **aSixtyFour** of the parent. Each **OccasionalType** element has a content including the attribute **@aRef** [Runapongsa, et. al, 2003]. The described information in this section together with the information given in the XML Schema are used to derive the semantics for performing the transformations.

As there is no modification made to the benchmark XML Schema, no data cleansing is performed. In the next section, data scaling is presented. As the names of all the hierarchy elements in this micro-benchmark schema are the same, i.e. eNest, this could raise a recursive schema problem. However, we have clarified earlier that the schema hierarchy's hieght is 16 which is determined by eLevel. Therefore the recursive problem has been avoided.

b. Data Scaling

To keep the micro-benchmark data set simple, we use two data sets of size 50 and 550 mega bytes (MB) given by the Data Generator developed by the micro-benchmark, which can be downloaded from [Michigan, 2011]. We use the large

document of 550 megabytes (MB) to scale down the data set (DS) including 0.2DS, 0.4 DS, 0.6 DS, 0.8 DS and 1SD. So, by adding 50 (MB) to the new scaled data sets, the complete sets of 50(MB), 150(MB), 250(MB), 350(MB), 450(MB) and 550(MB) is obtained.

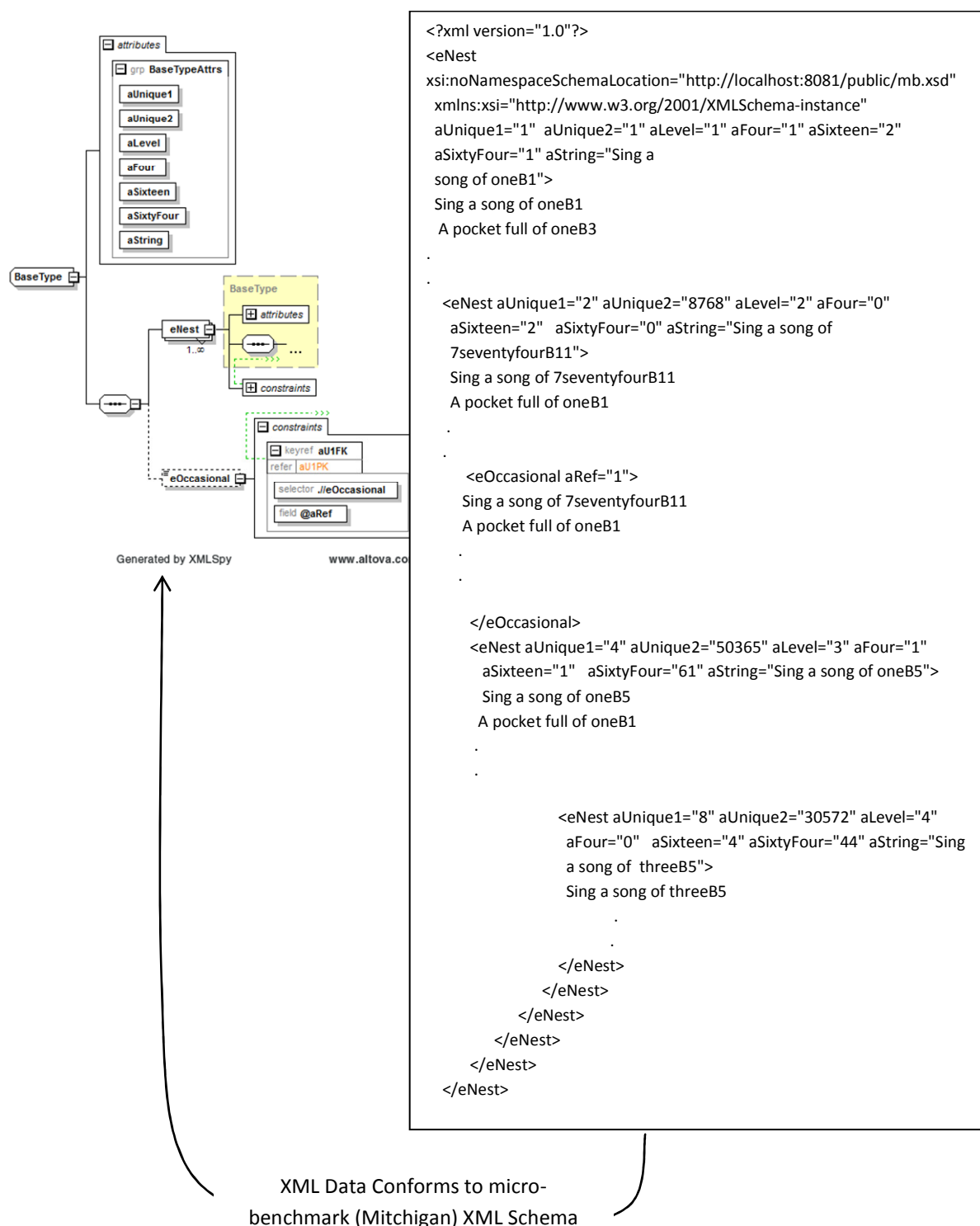


Figure 7.6 Micro-benchmark and Snapshot of XML Data

c. Query Selection

Unlike experiment 1, most of the XPath queries for experimentation are provided by the benchmark system. There are not many XPath query returning the sub-tree of an element, a few XPath queries are modified to select the sub-tree instead of the value of an element. A small number of XPath queries are newly created so that semantics such as subtype element can be applied for transformation.

For each semantic transformation category, we need to propose a set of selection XPath queries with regard to the query pattern and selection type. To make it simple, and due to the importance of XML structural selection, an XPath query either returns values of an XML element, or returns sub-trees.

- **Semantic Path Transformations**

Micro-benchmark XPath queries do not provide any query patterns suitable for the application of a semantic path complement. However, semantic path transformations are expected to have the following query pattern requirements:

- XPath query patterns are in a *descendant* hierarchy ‘//’ or *wildcard* element ‘*’. These are used to demonstrate a successful application of *Semantic Path Expansion*.
- XPath query patterns are in a *descendant* hierarchy (or optional operator ‘//’) or *wildcard* element ‘*’. These are used to demonstrate a successful application of *Semantic Path Contraction*.

- **Semantic Transformations for Axes**

The XPath queries designed and provided by a micro-benchmark system do not focus on XPath axes. The only axis that is incorporated in the XPath queries are the self ‘.’.

- **Semantic Transformations for Predicates**

Due to the richness in data hierarchies of the micro-benchmark data set, the XPath queries that support complex twig pattern queries will be addressed here as we are

not able to accomplish this by using DBLP data sets. The condition patterns accommodate hierarchies '/' and '//'.

- Structural condition type supports pattern '/' or '/' or axes which can enable parent-child or ancestor-descendant joins or a more complex one is twig join.
- Comparison values support constant value, range-value and path-value (full path expression based on key/keyref attribute).

We find that our choice of micro-benchmark data sets justify the evaluation purposes as their data characteristics and semantics complement each other, thereby producing an evaluation that is both complete and comprehensive.

7.4 Summary

In this chapter, we have described the design and the implementation of our experiment for evaluation. The chapter comprises three main sections:

- The first section describes the experimental design background and the strategy to achieve the experiment objective.
- The second section describes the expected strategies including framework, platform with minimal supportability, operational set-up such as hardware, software system modules including algorithm and system architecture that are shared by the two experiments.
- The third section presents the unique parts of the strategies used for each individual experiment. This includes the process and technique used to enhance semantics, cleanse data, scale data, design and select queries, metrics and computation procedures.

In the next two chapters, we present the evaluation results and analyses which enable us to identify the optimization devices.

Chapter 8

Experimental Evaluation - Using Real Data Sets

This chapter focuses on the experimental evaluation of semantic transformations using a real data set of DBLP which is accompanied by an XML schema. It is our goal to study the performance of XPath queries before and after undergoing semantic transformations. The query performance enables us to thoroughly evaluate the impact of semantic transformations that have been applied to XPath queries to obtain their semantic XPath queries, but more importantly, to identify semantic transformations as optimization devices.

8.1 Performance Evaluation Preface

As discussed in Chapter 7, each XPath query and its semantic XPath query over each data set would be executed for n runs. The execution time is accumulated for the last three runs. The average execution time is then produced based on the last three runs. The average execution time is referred to as the *performance result* throughout this chapter. While the performance result for the semantic XPath query is the average execution time plus the transformation time, the performance result for the original XPath query is the average execution time. The experimental evaluation is based on the performance results of the original XPath query and its semantic XPath query.

Readers are reminded that the constraints used to transform XPath queries are the information in lists Q and C , which have been derived from the XML schema and proposed in Chapter 4. While Q contains a list of unique paths in which each unique path is a full path of a sequence of schema elements that must start from the schema root. Finally C contains a list of constraints of the XML schema elements.

Based on the availability of semantics available in the DBLP schema, a set of XPath queries is designed to satisfy the semantic transformations. Due to the shallowness of the data hierarchies in the DBLP schema, hierarchy query types cannot be too expressive. This limitation can be overcome by exploring a benchmarking data set which is addressed in the next chapter. Moreover, XPath queries considered in this chapter are able to fully facilitate all semantic transformations.

An XPath query and its semantic XPath query are equivalent if and only if they produce the same result set even though they have different structures.

8.2 Semantic Path Transformation

This section presents a set of XPath queries that are transformed by applying semantic path transformations including *semantic path expansion*, *semantic path contraction* and *semantic path complement transformations*. Figure 8.1 presents a set of customized XPath queries. Each individual XPath query is presented with its relevant details such as Figure, Semantic Transformation, Path Pattern, Semantic XPath Query and Result Type.

a. Semantic Path Expansion Transformation

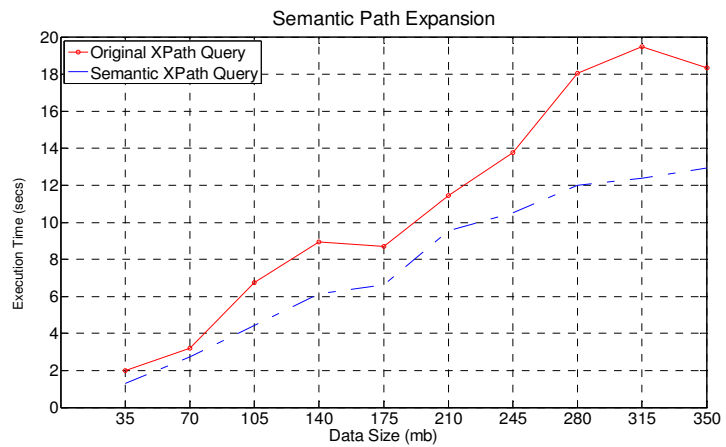
Figure 8.2 (a, b and c) shows the query performance results of XPath queries **//phdthesis//tn**, **dblp/inproceedings//tn** and ***/book/*/tn** and their semantic counterparts respectively. While the first two XPath queries use *descendant-ancestor* ‘//’ relationships, the third XPath query uses only wildcard ‘*’ and *parent-child* ‘/’ relationships. The *ancestor-descendant* ‘//’ relationship is purposely used in one XPath query and not in the other to observe its impact on query performance.

Figure	Semantic Transformations	Path Pattern	XPath query	Semantic XPath Query	Result Type
8.2 a	Path Expansion	Ancestor-descendant	//phdthesis//tn	dblp/phdthesis/title/tn	Values
8.2 b			dblp/inproceedings//tn	dblp/inproceedings/title/tn	
8.2 c		Wildcard	*/book/*/tn	phdthesis/book/title/tn	
8.3 a	Path contraction	Ancestor-descendant and wildcard	//*/isbn	//isbn	Sub-trees
8.3 b		Ancestor-descendant	*/*/title	//title	
8.3 c		Wildcard	dblp/*/title	//title	
8.4 a	Path Complement	Parent operator '..' and Ancestor-descendant	//book/year/../title/tn	//book/title/tn	Values
8.4 b		Parent operator '..'	dblp/phdthesis/title/tn/../tt	dblp/phdthesis/title/tt	

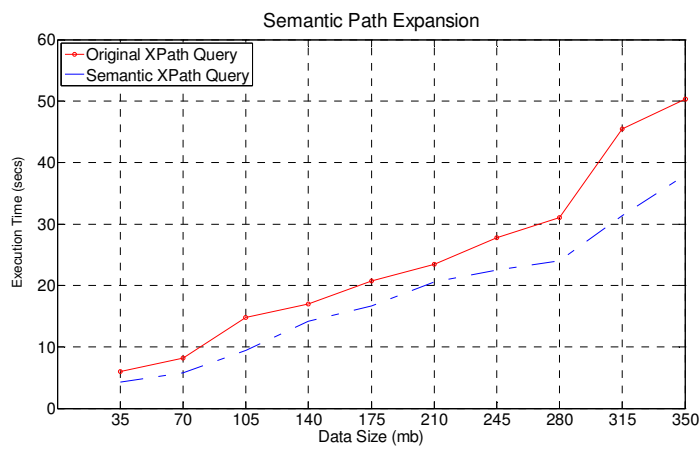
Figure 8.1 XPath Queries and Semantic XPath Queries by Semantic Path Transformation

For XPath query **//phdthesis//tn**, the semantic path expansion transformation replaces the path fragments **'//phdthesis'** and **'//tn'** with the path fragments **'dblp/phdthesis'** and **'title/tn'** based on the unique path **dblp/phdthesis/title/tn** located in list Q . For XPath query **dblp/inproceedings//tn**, semantic path expansion transformation replaces **//tn** with **title/tn** based on the unique path **dblp/inproceedings/title/tn** located in Q . For XPath query ***/book/*/tn**, the semantic path expansion transformation replaces ***/** with path fragment **'dblp/'** and **/*** with **/title/** based on the unique path **dblp/book/title/tn** located in Q . The performance results of semantic XPaths improve linearly along with the increase in the data sizes in Figure 8.2 (a and b). While the semantic XPath query gains an average of 40% over its original XPath query in Figure 8.2 (a); the semantic XPath query gains almost 22% over its original XPath query in Figure 8.2 (b).

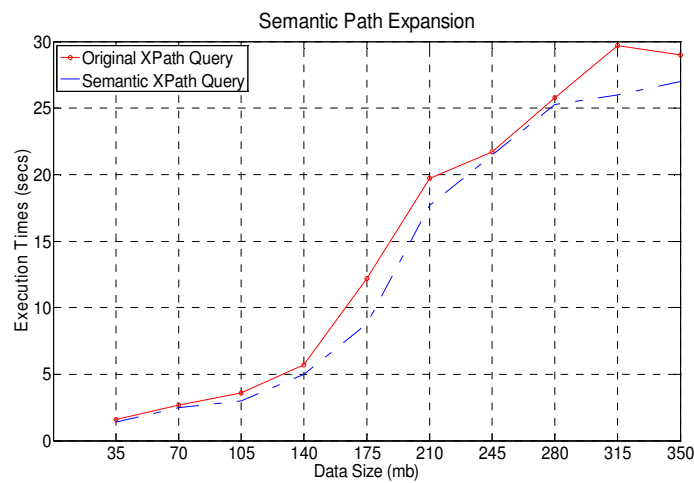
The overall performance achieved by the two semantic XPath queries in 8.2 (a) and 8.2 (b) is very good due to the fact that in the full path expressions, none of the elements is represented by **'*'**. In Figure 8.2 (c), the query performance results indicate that the semantic XPath query outperforms the XPath query by 4% to 9%. The performance result shows a slight linear improvement in performance as the data sizes increase.



(a)



(b)



(c)

Figure 8.2 Performance Results Before and After Semantic Path Expansion Applied

Although the first element in the XPath query in Figure 8.2 (c) starts with '*', the evaluation of the XPath query still starts from the root element **dblp**. This is because '*' is not led by the '/' relationship. The evaluation then searches for **book** as an immediate child of **dblp** where children of **book** are unknown, represented by '*'. Therefore, the evaluation continues to move down until it locates **tn**. To be able to search for all **tn** of **book**, it must repeat the same evaluation process. Due to the use of '/' in the XPath query and the semantic XPath query, the improvement between the original XPath query and the semantic XPath query is not significant in this case.

For the *descendant-ancestor* '/' and the *wildcard* '*' in the original XPath queries in 8.2 a, b & c, each is mapped to a single path fragment as each XPath query matches only one unique path. It also observes that the query with '/' performs worst than the query with '*' in any case. For such an XPath query pattern, semantic path expansion transformation is an optimization device if it can minimize the use of *descendant-ancestor* '/' and/or *wildcard* '*'.

b. Semantic Path Contraction Transformation

Figure 8.3 (a) depicts the performance results for XPath query **//*[@isbn]** and its semantic XPath query. The XPath query uses the '/' relationship and the wildcard '*'. The semantic path contraction transforms the XPath query by replacing the path fragment **//*[@]** with **/'** based on the unique paths **dblp/book/isbn**, **dblp/incollection/isbn** and **dblp/proceedings/isbn** located in *Q*. This is done for two reasons: (1) element **isbn** is the target element in the XPath query therefore any unique path that has **isbn** as its target element which has a valid parent element is picked up; and (2) the returned data set produced by XPath query **//*[@isbn]** is equivalent to the result produced by **dblp/book/isbn**, **dblp/incollection/isbn** and **dblp/proceedings/isbn**.

The graph in Figure 8.3 (a) shows that the performance results of the semantic XPath query grows linearly with the increase in data sizes. The growth of 20% for the larger data sets indicates that the bigger the data sizes, the greater the improvement on query performance will be.

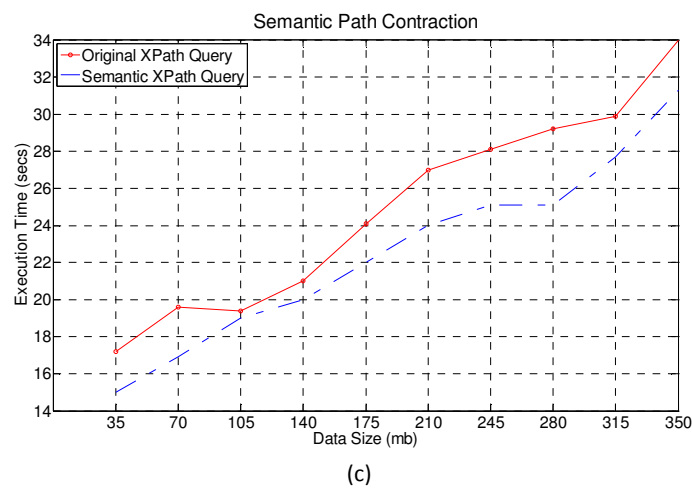
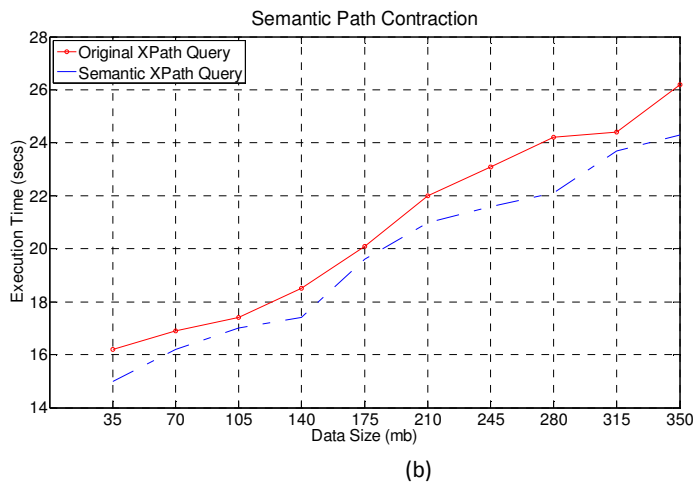
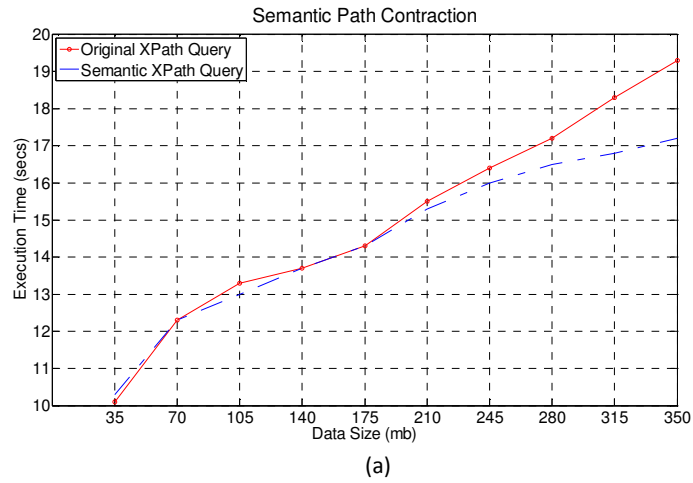


Figure 8.3 Performance Results Before and After Semantic Path Contraction Applied

The use of `/**` by the original XPath query `/**/isbn` means that for every hierarchical element, starting from the root, an element is evaluated for a descendant which must be a parent of `isbn`. Unlike the original XPath query, the semantic XPath query evaluates every element, that starts from the root, for a descendant that must

satisfy **isbn**. The evaluation process performed by the semantic XPath query is slightly less than the evaluation performed by the original XPath query.

Figure 8.3 (b & c) shows the performance result of XPath queries ***/**/title** and **dblp/**/title** and their semantic counterparts. Note that these two XPath queries use both wildcard '*' and hierarchy '/' before the descendant **title**. The semantic path contraction transforms XPath queries ***/**/title** and **dblp/**/title** by first locating unique paths that that has **title** and its further descendant due to the use of '/' preceding **titles**. The unique paths: **dblp/article/title/tn**, **dblp/article/title/tt**, **dblp/article/title/ref**, **dblp/inproceedings/title/tn**, **dblp/inproceedings/title/tt**,....., **dblp/www/title/tn**, **dblp/www/title/tt**, **dblp/www/title/ref** are located in list *Q*. These located unique paths contain descendants of **title** in the whole document. Due to the existing multiple unique paths, the semantic path contraction transformation then contracts the XPath queries to a single semantic XPath query, which is **//title**.

The performance of the semantic XPath queries in Figure 8.3 (b & c) show a constant improvement along with the growth in data sizes. Both graphs show a significant improvement of an average of 40% for smaller data sizes but a decline in improvement at a constant rate on an average of 25%, as the data size grow bigger.

Both XPath queries ***/**/title** or **dblp/**/title** have appeared to require the same evaluation number of elements such as **dblp**, **article**, **precedings**, **books**,..., **www** that occur before **title** due to the use of '*/'. While **dblp/**/title** appears to require less evaluation space due to the specified **dblp/***, XPath query ***/**/title** appears to require more evaluation space due to the specified ***/***. The first wildcard * in ***/**/title** does not make much difference in terms of evaluation space as it is identified as the root element **dblp** when evaluation begins.

By eliminating the wildcard '*/' in both XPath queries, a significant improvement would be expected as the data issue grows. However, this does not appear to do so; this could be due to the contracted path where XPath queries in both 8.2 (b & c) are contracted on the element type **title**. XPath query in 8.2 (a) is contracted on the leaf element. In the case of 8.2 (b & c), they both produce multiple semantic XPath queries. However, the overall performance of the semantic XPath queries is much better as shown in the graphs.

Based on our analyses and results, it can be concluded that semantic path contraction is an optimization device especially when it occurs on the leaf element.

c. Semantic Path Complement Transformation

Figure 8.4 (a & b) shows query performance results of XPath queries **//book/year/./title/tn** and **dblp/phdthesis/title/tn/./tt** as well as their semantic XPath query. Both XPath queries use parent operator ‘..’.

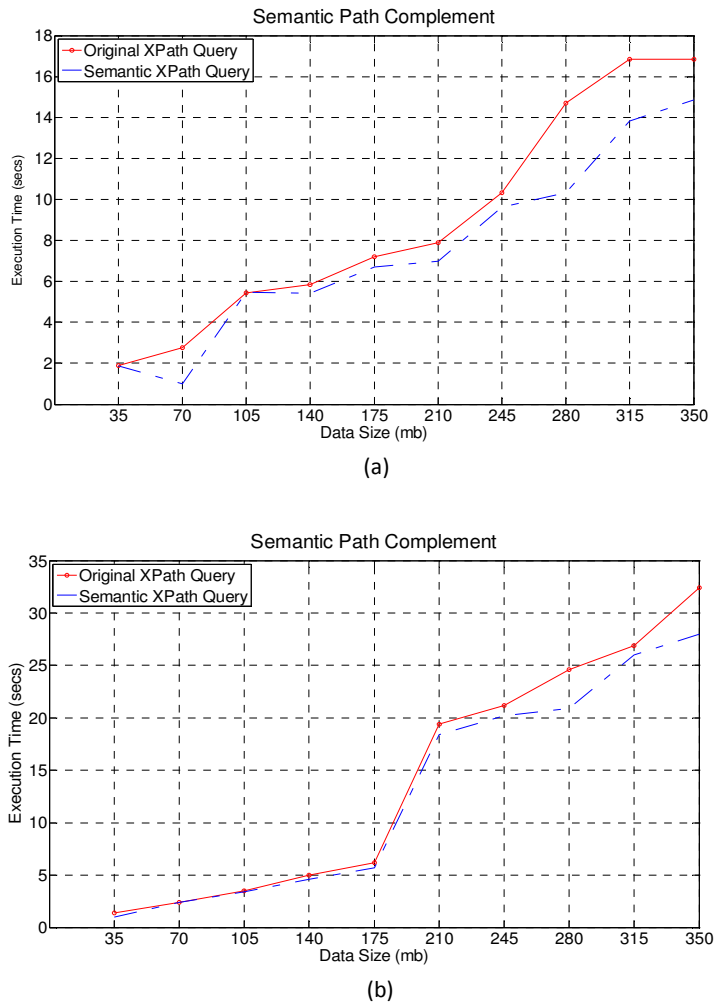


Figure 8.4 Performance Results Before and After Semantic XPath Complement Applied

The semantic path complement transforms the XPath queries by removing **year/./** and **tn/./** from the XPath queries **//book/year/./title/tn** and **dblp/phdthesis/title/tn/./tt** respectively.

The XPath query `//book/year/./title/tn` matches two unique paths **dblp/book/year** and **dblp/book/title/tn** where **book** is the branching element of **year** and **title**, in which **year** is the condition element.

The XPath query `dblp/phdthesis/title/tn/./tt` matches two unique paths **dblp/phdthesis/title/tn** and **dblp/phdthesis/title/tt** where **title** is the branching element (refer to Chapter 3 for the definition) of **tn** and **tt**, in which **tn** is the condition element. The semantic path complement transformation removes **tn/./** from **dblp/phdthesis/title/tn/./tt** and **year/./** from `//book/year/./title/tn` with respect to occurrence constraints of **year** and **tn** restricted in the schema that satisfy the semantic rule.

Both the query performances in Figure 8.4 (a & b) performed by semantic XPath query improves linearly along with the growth in data sizes. The use of parent ‘.’ requires the execution of an XPath query that is specified with a query condition element. The semantic path complements the transformation by removing the query condition elements from the XPath queries to reduce the processing of multiple path expressions. That is, the path expression from the root to the target element is considered as one path. The root expression to the query condition element is considered as another path. Hence, the execution of each XPath query is considered as the execution of two path expressions.

As shown in Figure 8.4 (a), the performance of the query indicates that the semantic XPath query outperforms the original query by between 2% and 8%. The rate of improvement on query performance continues to increase as the size of the data set grows.

Figure 8.4 (b) shows a fluctuating improvement on query performance. This is normal as there are only a small number of PhD thesis titles existing in the database. Nevertheless, when the information increases relatively in larger data sets, the query performance improvement actually increases as expected. As has also been observed, the execution times by XPath queries in Figure 8.4 (b) are more than those for 8.4 (a). This is probably due to a higher number of hierarchies in one XPath query than in the other.

Based on the result patterns in Figure 8.4 (a & b), it can be concluded that the semantic path complement transformation is an optimization device with regards to two noticeable facts: 1) the greater the amount of data present for retrieval, the greater the improvement on query performance achieved by the semantic path complement query will be and 2) when more query hierarchies appear in the XPath queries, the longer it will take longer for traversal.

8.3 Semantic Transformations for Predicates

In this section, given XPath queries are transformed by applying *predicate elimination* and *predicate reduction semantic transformations*.

a. Predicate Elimination Semantic Transformation

Figure 8.6 (a) shows the query performance results of the XPath query **dblp/phdthesis[supervisor=/dblp/ phdthesis/@key]/title/tn** and its counter-part. The predicate contains a single condition of an element that has a key reference attribute constraint. The comparison element **supervisor** has a value of a path expression **/dblp/ phdthesis/@key** that has only a parent-child relationship. In order to remove the predicate, the query condition **supervisor=/dblp/ phdthesis/@key** must be determined with a full-qualifier status.

As the query condition status determination rule locates the matched unique path **dblp/phthesis/supervisor** in list Q , and **dblp/phthesis/supervisor = dblp/phthesis/@key** in list C that allows a full-qualifier status to be awarded to the condition. To remove the predicate, the predicate reduction semantic transformation then verifies that the **supervisor** element satisfies the semantic transformation rules. In this query, the minimal occurrence must be 1 or above and the query join condition is NULL.

Figure	Query Type	Join	XPath query	Semantic XPath Query	Returned Results
8.6 a	Query Condition with Path Value	None	dblp/phdthesis[supervisor=/dblp/phdthesis/@key]/title/tn	dblp/phdthesis/title/tn	Values of Leaf Node/Attributes
8.6 b	Query Condition with No Value		dblp/proceedings[title/tn]/url	dblp/proceedings/url	
8.6 c	Query Condition with Range Value		dblp[proceedings/year>=1950]/inproceedings/title/tn	dblp/inproceedings/title/tn	
8.7 a	Connective Query Conditions with Range Value	AND	dblp/incollection[author and year >=1950]/title/tn	dblp/incollection/title/tn	
8.7 b	Connective Query Conditions with Range Value		dblp/incollection[title and year >=1950]/title/tt	dblp/incollection/title/tt	
8.8 a	Connective Query Condition with Attribute and Range value		//book[@mdate and title/tn and year >=2000]/@key	//book[year>=2000]/@key	
8.8 b	Connective Query Condition with Attribute and Range value		//book[@mdate and title/tn and year >=2000 and author]/@key	//book[year>=2000]/@key	
8.9 a	Connective Query Conditions with .Matching Value & Range Value	OR	dblp/book[year=1948 or year<=2010] /author	dblp/book[year<=2010]/author	
8.9 b	Connective Query Conditions with Range Value		dblp/book[year < 1950 or url]/title/tn	dblp/book[url]/title/tn	

Figure 8.5 Original and Semantic XPath Queries and Related Information

Figure 8.5 summarizes a set of XPath queries, including the figure numbers where the query performance is provided, their query types, types of query conditions joins in the predicates, the corresponding semantic XPath query and the result selection type.

The query performance result in Figure 8.6 (a) increases linearly along with the increase of data sizes. The query performance of semantic XPath query is effectively improved by between 20% and 25% on larger data sets of 210 to 350 megabytes. The query result pattern for the smaller data sets shows an improvement. From the improvement pattern in the larger data set, it appears that the key reference element

does not perform effectively on smaller data sets. It can be concluded that semantic transformation for predicate reduction for this query pattern is an optimization device.

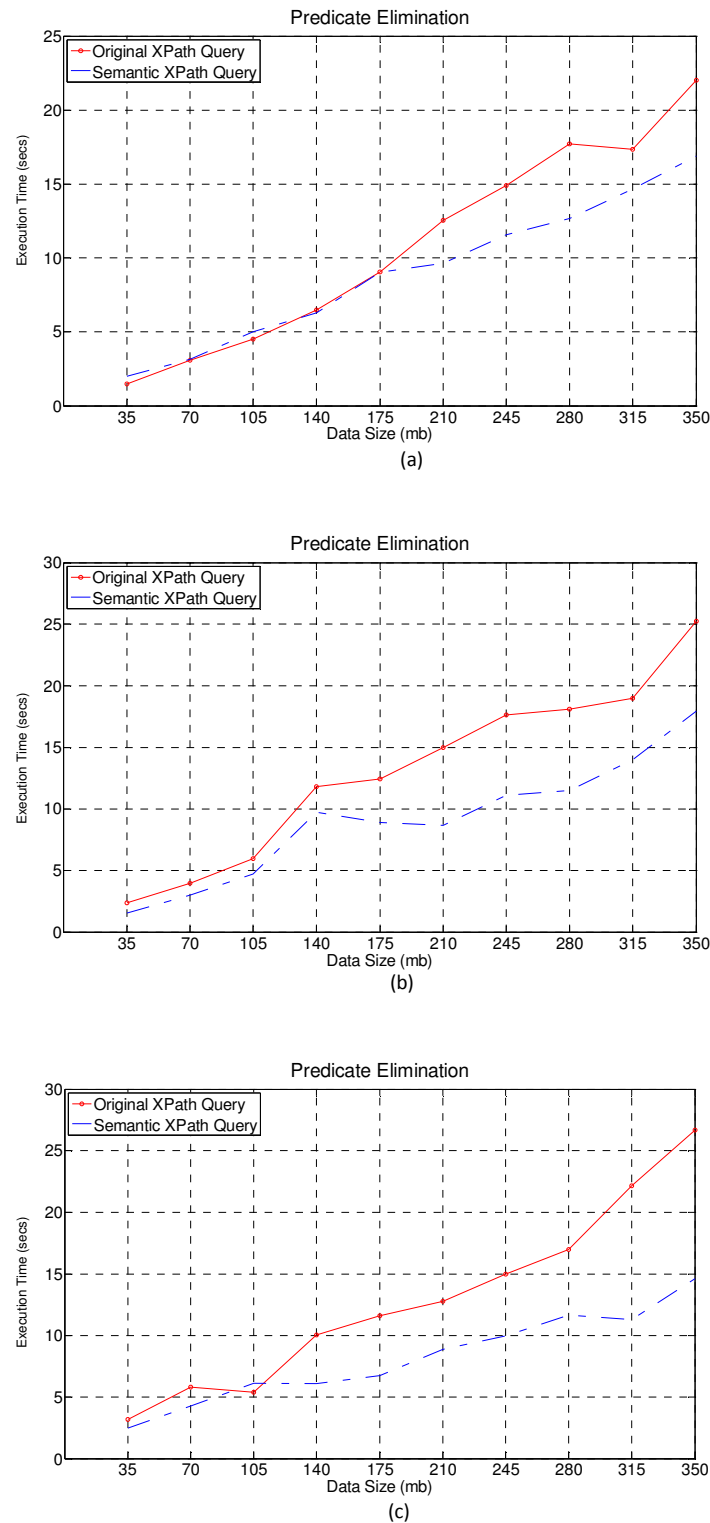


Figure 8.6 Performance Results Before and After Predicate Elimination Applied

Figure 8.6 (b & c) show the performance results of XPath queries **dblp/proceedings[title/tn]/url**, **dblp/proceedings/year>=1950]/inproceedings/title/tn** and of their semantic XPath queries. Both XPath queries demonstrate a single query condition using a path fragment. While the first query condition **title/tn** has no comparison value, the second query condition **title/tn** has a comparison value of **1950**.

The query condition status determination rule locates the matched unique paths **dblp/proceedings/title/tn** and **dblp/proceedings/url** in list *Q* and **proceedings/year** 1950 2020 in list *C*. Such located information gives both query conditions full-qualifier status.

To remove the predicates, the predicate reduction semantic transformation then verifies whether the query conditions **title/tn** and **proceedings/year >= 1950** satisfy the semantic transformation rules. List *C* shows that the minimal occurrence of **title** and **year** under **proceedings** is 1 and the published **year** is between 1950 and 202 as well as that the query join condition is NULL. Hence, the predicates have been successfully removed from the XPath queries.

As shown in Figure 8.6 (b & c), the query performance improvement for both semantic XPath queries increases linearly along with the growth in the size of data sets. The average gain improvement on query performance is between 25% and 50%. Hence it can be concluded that the predicate reduction semantic transformation, applied to remove a predicate when its query condition is a path fragment, is an optimization device.

The improvement of query performance achieved by semantic XPath queries can be explained on follows: for all the semantic XPath queries in 8.6, the evaluation starts from the root that has children which must satisfy specific elements before the next children are evaluated. Due to the hierarchy ‘/’ used in all semantic XPath queries, the evaluation is able to minimize the search space. With the original XPath queries, the hierarchy element in which the predicate is specified is evaluated before the hierarchy element in the path is evaluated.

For instance in 8.6 (a) the evaluation of XPath query starts from the root **dblp** that has children which must satisfy element **phdthesis** where the predicate **[supervisor=/dblp/phdthesis/@key]** is specified. The **phdthesis** is evaluated for children that must satisfy element **supervisor**, which must also satisfy a referenced value for **dblp/phdthesis/@key**. In 8.6 (b) the predicate **[title/tn]** does not have a comparison value and the evaluation needs to assure the condition path is valid **title/tn** before the next hierarchy element **url** in the main path is evaluated. Therefore, the required evaluation of conditions and the path condition in the XPath queries takes much longer to process.

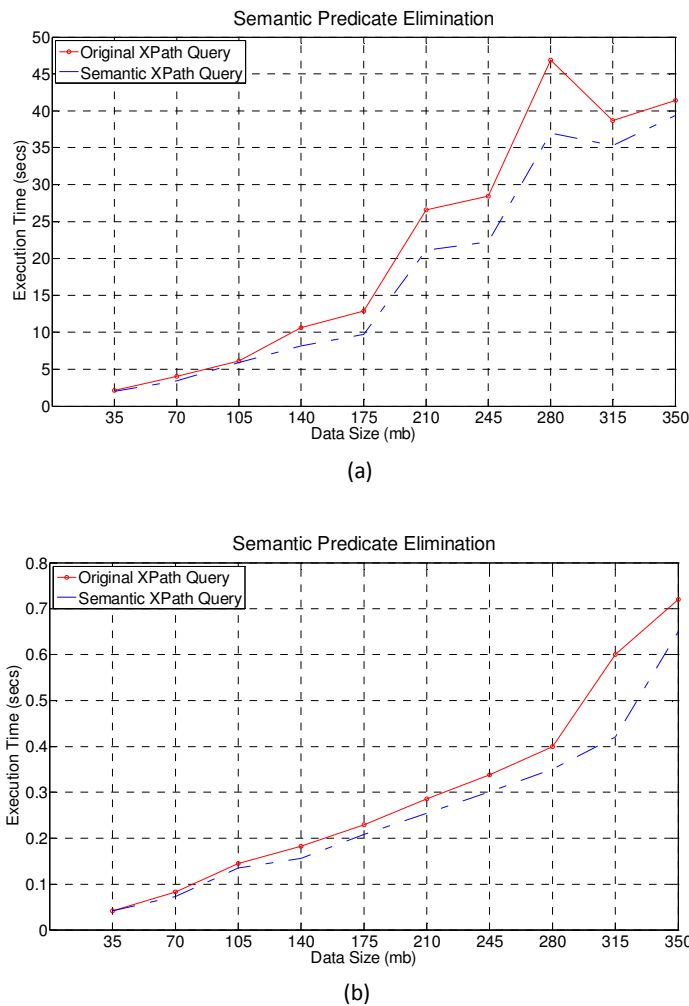


Figure 8.7 Performance Results Before and After Predicate Elimination Applied

Figure 8.7 (a & b) shows the query performance results of XPath queries **dblp/incollection[author and year >=1950]/title/tn**, **dblp/incollection[title and year >=1950]/title/tt** and of their semantic counterparts. The XPath queries

demonstrate the use of the AND operator between the query conditions. In order to remove the predicate, each condition must be verified as a full-qualifier.

Predicate reduction semantic transformation eliminates the predicate **[author and year >=1950]/title/tn** from the XPath query **dblp/incollection[author and year >=1950]/title/tn** with the regards to the matching unique paths **dblp/incollection/author**, **dblp/incollection/year** and **dblp/incollection/title/tn**. In addition, elements **author** and **year** have occurrences that satisfy the semantic transformation rule. The range values of element **year** must fully match the range values of the year element under **collection** specified in the XML Schema.

The predicate reduction semantic transformation is also able to remove the predicate in the XPath query **dblp/incollection[title and year >=1950]/title/tt** with regards to the matching unique paths **dblp/incollection/title**, **dblp/incollection/year** and **dblp/incollection/title/tt**. Based on the information located in list C, elements **title** and **year** have occurrence that satisfies the semantic transformation rule and the range value of element **year** fully matches the range value of **year** under **collection** specified in the XML Schema.

The query performance improvement grows linearly along with the growth in data sizes in Figure 8.7 (both a & b). This means that the overall performance of the semantic XPath queries is better than that of the XPath queries. However, due to the simple condition and the insignificant amount of existing data in the DBLP database, semantic XPath queries do not show a significant improvement. Nevertheless, there is some evidence of improvement in query performance based on the graph patterns shown in Figure 8.7.

b. Predicate Reduction Semantic Transformation

This section demonstrates the predicate reduction semantic transformation that reduces the size of a predicate by removing some conditions which are identified as full-qualifiers.

Figure 8.8 (a & b) shows the query performance the XPath queries **//book[@mdate and title/tn and year >=2000]/@key**, **//book[@mdate and title/tn and year**

>=2000 and author]/@key and of their semantic counterparts. The query conditions in the predicate demonstrate the use of the AND operator.

The predicate reduction semantic transformation, based on the identified unique paths, determines the full-qualifiers for query conditions. This is because **@mdate** has a *required* attribute value, and both **title/tn** and **author** have an *occurrence* constraint that satisfies a minimal occurrence of 1 as required by the semantic rule proposed in Chapter 5. Therefore, **@mdate**, **title/tn** and **author** have been removed from the predicates.

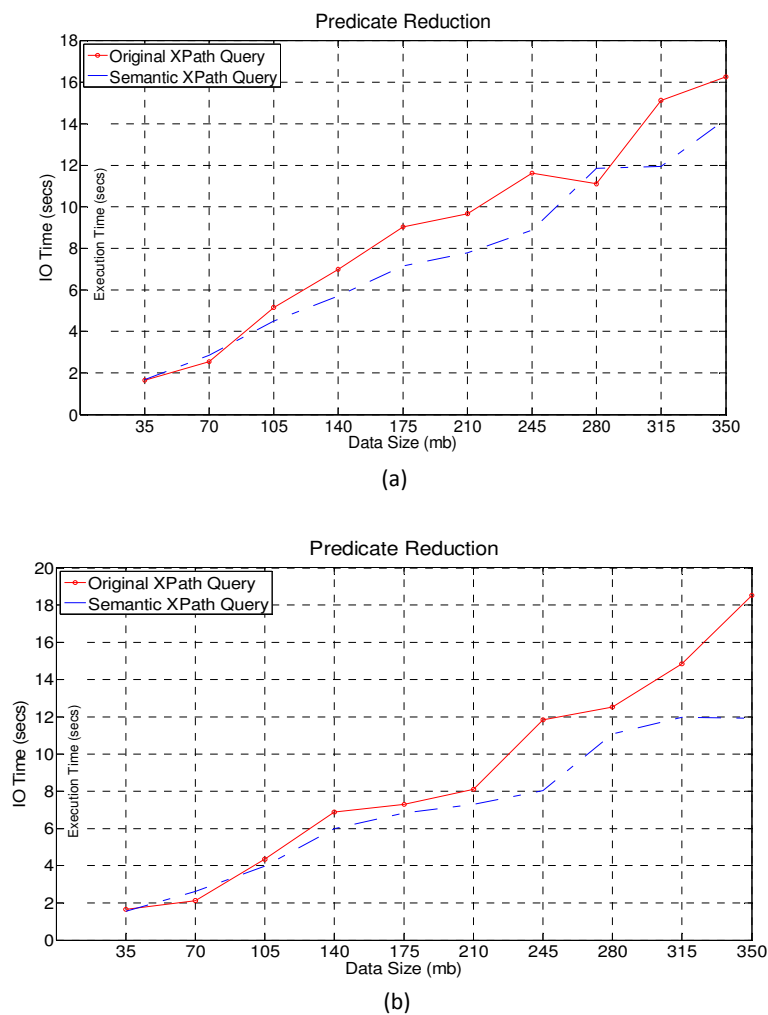


Figure 8.8 Performance Results Before and After Predicate Reduction Applied

As shown in Figure 8.8 (a and b) the semantic XPath queries outperform the original XPath queries between 2% and 20%. The query performance for both the XPath query and the equivalent semantic XPath query shows a linear pattern along with the growth of data sizes. The semantic XPath queries outperform the original XPath

queries due to reduced evaluation of the query conditions as **@mdate**, **title/tn** and **author** have been removed from the XPath queries after performing the transformation.

The slowness of query performance shown by the original XPath queries can be caused by the fact that each query condition is evaluated along the path that must start from the root of tree. For example, XPath query **//book[@mdate and title/tn and year >=2000]/@key** has two query conditions **@mdate and title/tn and year >=2000**, therefore the evaluation must start from the root of the document which is **dblp** until it finds the descendant **book**, then it continues to search for **@mdate**. To evaluate **title/tn**, it starts again from the root of the document tree which is **dblp** and traverses down until it finds **title/tn** under **book**. Once the conditions are satisfied, the **book** is then evaluated which must be started from the root of document tree **dblp**, for **@key**.

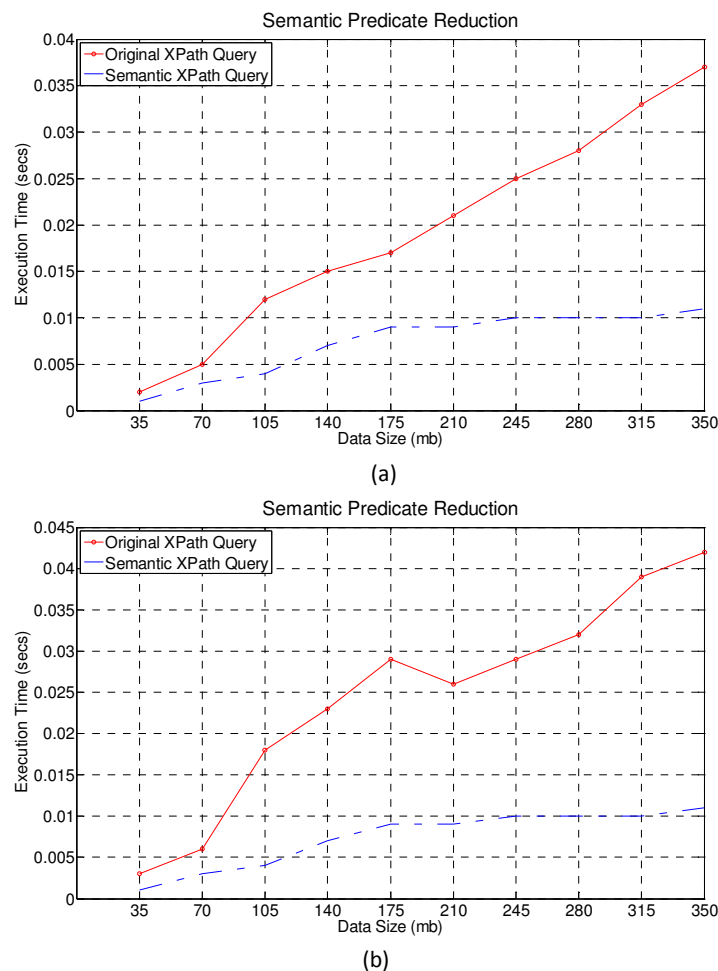


Figure 8.9 Performance Results Before and After Predicate Reduction Applied

Graphs in Figure 8.9 (a & b) depict the query performance results of XPath queries **dblp/book[year=1948 or year<=2010]/author**, **dblp/book[year < 1950 or url]/title/tn** and their semantic counterparts. The XPath queries demonstrate the use of OR between the query conditions.

To confirm the correctness of path expressions in XPath queries **dblp/book[year=1948 or year<=2010]/author** and **dblp/book[year < 1950 or url]/title/tn**, semantic transformation uses matching unique paths **dblp/book/year**, **dblp/book/author**, **dblp/book/url** and **dblp/book/title/tn** in list *Q*. For **dblp/book[year=1948 or year<=2010]/author** the semantic transformation first verifies the query conditions **year=1948** and **year < 1950** and confirms that they are mutually inclusive because **year=1948** is never present in the database because the *inclusive* constraint set for **year** is between 1950 and 2020. Query conditions **year<=2010** and **url** are partial-qualifiers as the specified value 2010 is within the range values of the **year** and the **url** has a minimal value of 0 for the occurrence constraint found in list *C*.

Due to the use of OR between two query conditions in [**year=1948 or year<=2010**], the mutually inclusive query condition exists because **year = 1948** is in conflict with what was found for the years between **1950** and **2020** in list *C*. The mutual inclusive query condition does not cause the whole XPath query to return an empty answer, but allows the co-existence of data produced by the other query condition **year <=2010**.

The result patterns in Figure 8.9 (a & b) show a linear growth along with the growth in data sizes. With the increase in data sizes, the rate of improvement of between 50% and 85% achieved by semantic XPath queries also increases significantly. This shows that the larger the data sizes, the better the improvement of query performance for the semantic XPath queries will be.

The transformation of an XPath query using AND between the query conditions is very different from those using OR. For OR between query conditions, as soon as it is verified that one condition is a full-qualifier, the semantic rule can immediately remove the condition. Whereas if AND is used between the query conditions, the query condition cannot be removed even though it is verified and given a full-

qualifier; all query conditions are completely verified and then the query condition is removed based on its status.

8.4 Semantic Transformation for Axes

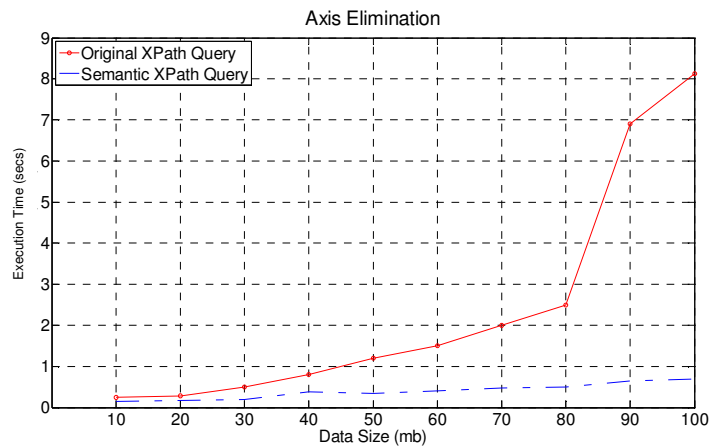
This section demonstrates the semantic transformations for the XPath query that are specified with XPath axes. XPath query performance specified with XPath axes is evaluated to identify opportunities for semantic transformations. The family of XPath axes includes *child*, *descendant*, *descendant-or-self*, *parent*, *self*, *following*, *following-sibling*, *preceding*, *preceding-sibling*, *ancestor* and *ancestor-or-self*. Figure 8.10 summarizes a list of XPath queries used to perform the evaluation.

Figure	XPath Query	Semantic XPath Query	Returned Result
8.11a	dblp/article/preceding-sibling::*	dblp/article[position() < last()]	Sub-tree
8.11b	dblp/article/following-sibling::*	dblp/article[position() > 1],dblp/inproceedings,dblp/proceedings,dblp/book,dblp/phdthesis,dblp/incollection,dblp/www	
8.12a	dblp/article/title/preceding::*	dblp/article[position() < last()],dblp/article[position() = last()]/author	Values of attributes and leaf nodes
8.12b	dblp/inproceedings/title/following::*	dblp/inproceeding[position() > 1],dblp/inproceeding[position() = 1]/pages,dblp/inproceeding[position() = 1]/year,dblp/inproceeding[position() = 1]/url,dblp/proceedings,dblp/book,dblp/incollection/dblp/phdthesis,dblp/www	Values of attributes, leaf nodes and sub-trees
8.13a	dblp/*/title/ancestor::*	dblp,dblp/article,dblp/inproceedings,dblp/proceedings,dblp/book,dblp/incollection,dblp/phdthesis,dblp/www	Sub-trees
8.13b	dblp/*/ancestor-or-self::*	dblp,dblp/article,dblp/inproceedings,dblp/proceedings,dblp/book,dblp/incollection,dblp/phdthesis,dblp/www	
8.14a	//inproceedings[@key]/title[tn]/preceding-sibling::*	//inproceedings/author	Values of attributes and leaf elements
8.14b	//article[@key]/title[tn]/preceding::*	//article[position() = last()]/author //article[position() < last()]	

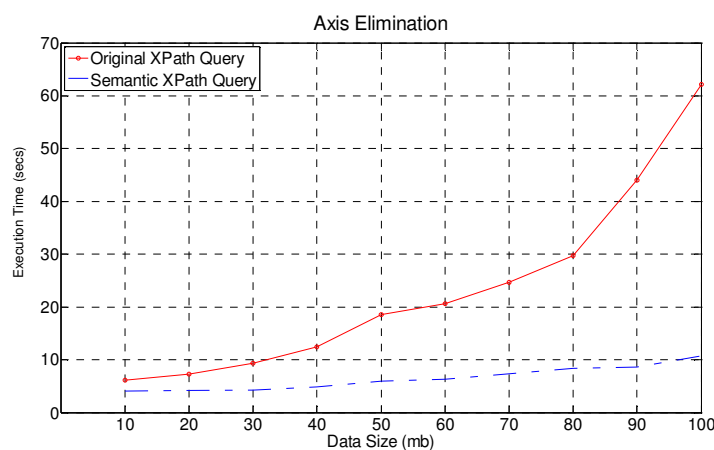
Figure 8.10 Original and Semantic XPath Queries and Related Information

Figure 8.11 (a & b) shows the query performance result of XPath queries **dblp/article/preceding-sibling::***, **dblp/article/following-sibling::*** and their semantic counterparts. The XPath queries demonstrate the use of *preceding-sibling* and *following-sibling* axes.

The semantic transformation removes the *preceding-sibling* and *following-sibling* axes from the XPath queries by following the semantic rule proposed in Chapter 5. The *preceding-sibling* and *following-sibling* XPath queries retrieve information about siblings that occur before or after the last or first occurrence of the context element.



(a) Before and After Removing Preceding-sibling Axis



(b) Before and After Removing Following-sibling Axis

Figure 8.11 Before and After Semantic Transformation for Preceding-/Following-sibling Axis Applied

The unique paths listed before **dblp/article** whose context element must be the sibling of **article** and after **dblp/article** whose context element must be the sibling of **article** in list Q , are the target semantic XPath queries. XPath query selects the information of the context element that will be added to the target semantic XPath queries if it has an *occurrence* constraint in which the minimal occurrence is 1 or above and the maximal occurrence is above 1.

For the XPath query **dblp/article/preceding-sibling::*** there are no unique paths to produce siblings that precede the **article**. When the *occurrence* constraint of **article** has an occurrence between 1 and many, the unique path selects the ‘all’ **article**, the transformation would then:

- add the context function [position()<last()] to the **article** so that the last occurrence of **article** is not selected.

For the XPath query **dblp/article/following-sibling::*** unique paths such as **dblp/inproceedings**, **dblp/proceedings**, **dblp/book**, **dblp/phdthesis**, **dblp/incollection**, **dblp/www** are the semantic XPath queries because they are listed after the unique path that selects the articles. In addition, the *occurrence* constraint of **article** has an occurrence between 1 and many; unique path selects the information of **article** and this must be added to the semantic XPath queries. The transformation would:

- add the context function [position()=1] to the **article** so that the first occurrence of **article** is not selected.

See Figure 8.10 for the full list of the semantic XPath queries.

The query performance results in Figure 8.11 (a & b) indicate an *exponential curve* along with the growth in the data sizes with the *original* XPath queries, but a *linear* growth along with the growth in the data sizes with the *semantic* XPath queries. In both 8.11 (a & b) figures, the semantic XPath queries significantly outperform the original XPath queries by between 50% and 80%. The difference in query performance between the XPath queries and their semantic XPath queries indicates a significant query improvement.

As mentioned earlier, semantic transformation removes the axis from the XPath query to support some commercial database systems which are unable to support the processing of XPath query axes. One of the most outstanding commercial database systems (For lisencing reason, the name of the database can not be disclosed) is adopted to carry out the experimental evaluation; hence, the semantic XPath queries significantly outperform the original XPath queries.

Figure 8.12 (a & b) shows the performance results of the XPath queries **dblp/article/title/preceding::***, **dblp/inproceedings/title/following::*** and their semantic XPath queries. The XPath queries demonstrate the use of *preceding* and *following* axes.

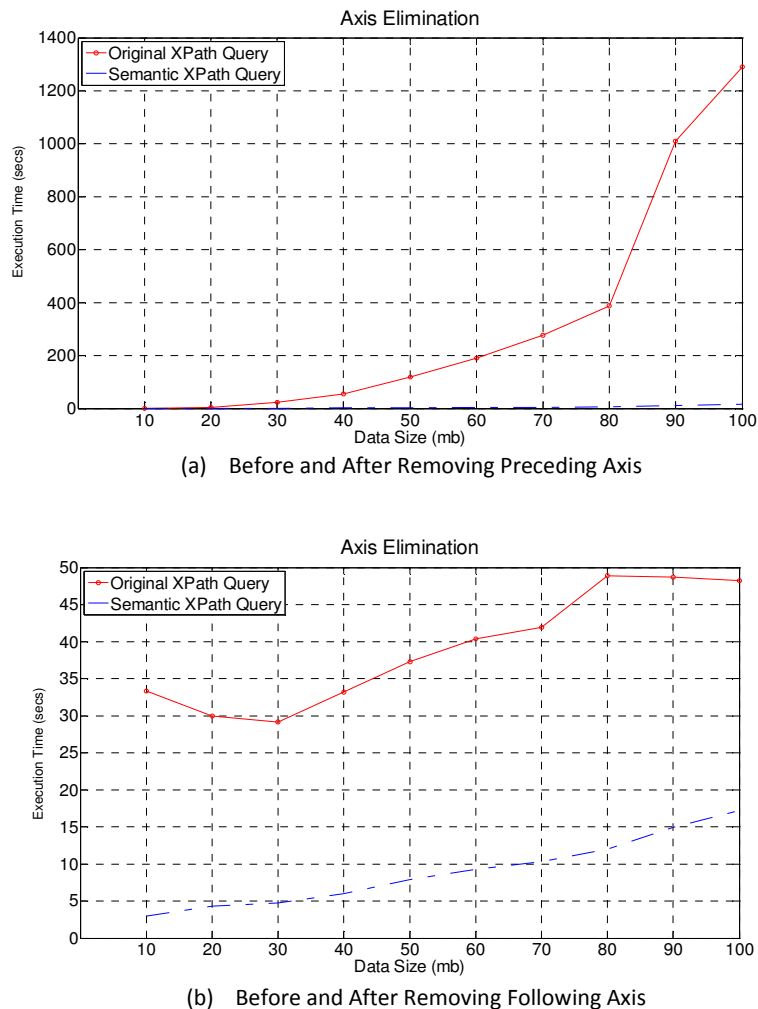


Figure 8.12 Before and After Semantic Transformation for Preceding/Following Axis Applied

The semantic transformation removes the *preceding* and *following* axes from the XPath queries by following the semantic rule proposed in Chapter 5. The *preceding*

and *following* XPath queries retrieve information that occurs before or after the last or the first occurrence of the context element including the ancestors or descendants.

Due to the efficiency of the derivation of our proposed unique paths described earlier in Chapter 4, the unique paths listed before **dblp/article/title** and those after **dblp/inproceedings/title** in list Q are the target semantic XPath queries. XPath query selects the information of context element which will be added to the target semantic XPath queries if it has an *occurrence* constraint in which the minimal occurrence is 1 and above, and the maximal occurrence must be above 1.

For XPath query **dblp/article/title/preceeding::***, unique paths such as **dblp/article/author** is the semantic XPath query, because it is listed before the unique path that selects the title of the article. In addition, the *occurrence* constraint of the article title has an occurrence between 1 and many, so the unique path selects the information of article title that must be added to the semantic XPath queries. The transformation would:

- add the context function [position()<last()] to the unique path **dblp/article** so that the last occurrence of **article** title in the whole document is not selected.
- add the context function [position()=last()] to the **author** in unique path **dblp/article/author** so that all authors under the last article are selected; this is because the **author** occurs before **article title** in the XML document.

For XPath query **dblp/inproceedings/title/following::***, unique paths such as **dblp/proceedings**, **dblp/book**, **dblp/incollection/dblp/phdthesis**, **dblp/www** are the semantic XPath queries because they are listed after the unique path that selects the **inproceedings** title. In addition, since the *occurrence* constraint of **title** of **inproceedings** has an occurrence between 1 and 1, the unique path selects the information of **title** of **article** that must be added to the semantic XPath queries; hence, the transformation would:

- add [position()>1] to element **inproceedings** in **dblp/inproceeding** so that it can select all information of **inproceedings** except the first occurrence of **inproceedings**

- add [position()=1] to element **inproceedings** in **dblp/inproceeding/pages**, **dblp/inproceeding/year**, **dblp/inproceeding/url** to select all information, occur after title, of the first occurrence of **inproceedings**

The query performance result patterns shown in Figure 8.12 (a) indicate an exponential growth along with the growth of the data sizes by the original XPath query, and a linear growth along with the growth in the data sizes by a set of semantic XPath queries. It has been observed that the total time taken by the semantic XPath queries is significantly smaller compared with the time taken by the original XPath query, with the performance improvement percentage being between 95% and 100%.

As can be seen, the query performance result patterns in Figure 8.12(b) increase linearly along with an increase in the data sizes for both the original and the semantic XPath queries. The performance by the semantic XPath queries shows a consistent improvement rate (almost 90%), along with the growth in the data sizes.

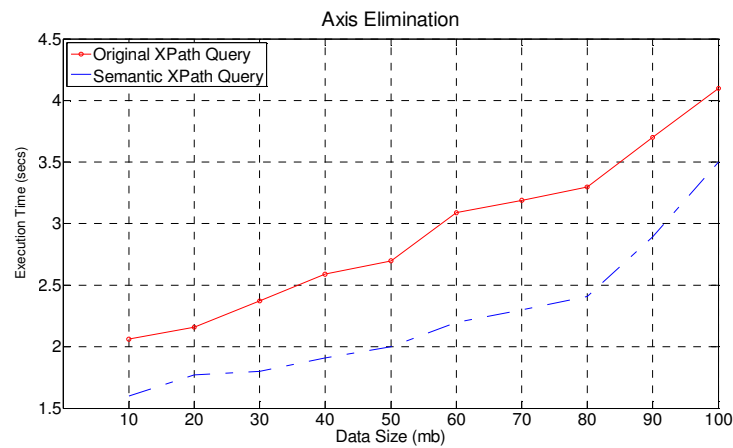
As demonstrated in Figure 8.12 the query performance of XPath queries specified with *preceding* or *following* axis are significantly slow as the data sizes increase, but query performance performed by semantic XPath queries increases linearly. The semantic transformations are considered as optimization devices especially for XML-enabled database management systems that inadequately support XPath axes.

Figure 8.13 (a & b) shows the query performance results of the XPath queries **dblp/*/title/ancestor:***, **dblp/*/title/ancestor-or-self:*** and their semantic XPath queries. These XPath queries demonstrate the use of *ancestor* and *ancestor-or-self* axes.

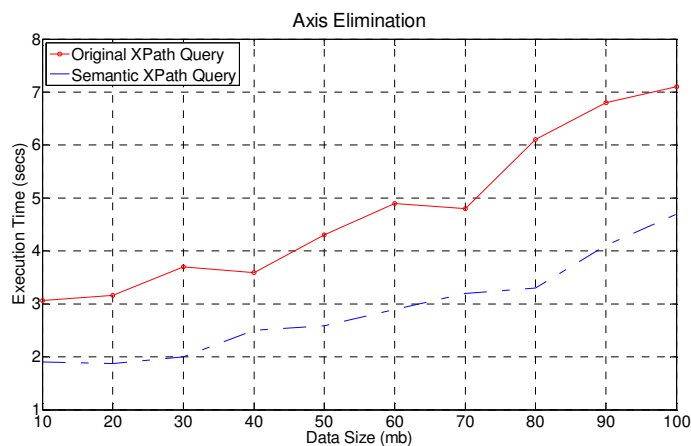
The XPath queries specified with the *ancestor* axis selects information about all the ancestors of the context element. The semantic transformation locates all the unique paths that are listed before the one that selects the information of the context element. For the XPath query **dblp/*/title/ancestor:***, the unique paths **dblp**, **dblp/article**, **dblp/inproceedings**, **dblp/proceedings**, **dblp/book**, **dblp/incollection**, **dblp/phdthesis** and **dblp/www** are the semantic XPath queries. They are listed before all the unique paths that select information about **title** in list *Q*. Due to the use

of ‘*’ between **dblp** and **title** elements in the XPath query, further semantic XPath queries are produced.

The XPath queries specified with the *ancestor-or-self* axis select information about all the ancestors of the context element. The semantic transformation locates all the unique paths that are listed before the one that selects the information about the context element. For the XPath query **dblp/*/title/ancestor-or-self::***, unique paths **dblp/article**, **dblp/inproceedings**, **dblp/proceedings**, **dblp/book** **dblp/incollection** **dblp/phdthesis**, **dblp/www**, **dblp/article/title**, **dblp/inproceedings/title**, **dblp/proceedings/title**, **dblp/book/title** **dblp/incollection/title**, **dblp/phdthesis/title**, **dblp/www/title** are the semantic XPath queries. They are listed before all unique paths that select information of **title** in list Q .



(a) Before and After Removing Ancestor Axis



(b) Before and After Removing Ancestor-or-self Axis

Figure 8.13 Before and After Semantic Transformation for Ancestor/Ancestor-Or-Self Axis Applied

The query performance result patterns in Figure 8.13 (a & b) show a linear growth along with an increase in data sizes. The semantic XPath queries in Figure 8.13 (a) show a significant query performance improvement of almost 80% initially. Unfortunately, they show a smaller query performance improvement rate of about 30% when approaching larger data sizes. This may be caused by the irregular amount of data being retrieved or some hiccups during the processing stage. On the other hand, the semantic XPath queries in Figure 8.13 (b) show a constant query performance improvement rate of between 40% and 50%. This is because the ancestor-or-self axis can be optionally specified by the operator ‘//’ in some XML-enabled database management systems. There may also be an interpretation between the axis and the operator done by the database engine which is not disclosed to the public by the vendor. This research is unable to comment on such information.

For a query specified with an *ancestor-or-self* axis, the performance improvement will be reflected at a consistent rate along with the growth in the data sizes. However, for the XPath query specified with an *ancestor* axis, further investigation is planned for future work. Either the query processing foundation of the adopted database needs further study, or a benchmark needs to be run on those queries on which the current semantic transformations cannot perform well.

Figure 8.14 (a & b) show the query performance results of XPath queries **//inproceedings[@key]/title[tn]/preceding-sibling::***, **//article[@key]/title[tn]/preceding::*** and their semantic XPath queries.

The XPath query **//inproceedings[@key]/title[tn]/preceding-sibling::*** demonstrates the use of the XPath axis *preceding-sibling* and also of predicates. The semantic transformation locates the unique paths that are listed preceding the unique path selects title of proceedings. The located unique path must have a context element that is a sibling of **title**, and in this case **dblp/inproceedings/author** is located. In addition, the *occurrence* constraint of **title** of **inproceedings** has an occurrence between 1 and 1, and apart from **author** as the sibling, **title** also has other **title** as a sibling. Therefore **title** of **inproceedings**, except the last **title** of **inproceedings**, must also be selected. Because each **inproceedings** has only 1 **title**,

only the unique path that selects **author** of **inproceedings** would be considered to be semantic.

As the XPath query is also specified with predicates **[@key]** and **[tn]**, semantic transformation for predicates has been applied to remove both the predicates. This is due to the required value of attribute of **key** and occurrence constraint of **tn**, which has an occurrence between 1 and 1. Both constraints confirm the existence of **inproceedings** and **title** in the database.

The XPath query `//article[@key]/title[tn]/preceding::*` demonstrates the use of the XPath axis *preceding* and also of predicates. The semantic transformation locates the unique paths that are listed in front of the one that selects **title** of proceedings. The located unique path must have a context element that is a sibling of **title** as well as those that occur in front of **title**; in this case, unique paths are **dblp/article/author** and **dblp/article**. However, to make the two unique paths produce a result equivalent to the result produced by the original XPath query, semantic transformation needs to verify the *occurrence* constraint of article title. The **title** has occurrence between 1 and 1, so the semantic transformation would:

- add the context function `[position()<last()]` to the **article** in unique path **dblp/article** so that the last occurrence of **article** is not selected
- add the context function `[position()=last()]` to the **article** in unique path **dblp/article/author** so that the last occurrence of **author** is selected

The patterns of performance results in Figure 8.14 (a) show a linear increase along with the increase in data sizes. The semantic queries outperform the original XPath queries by a percentage between 95% and 98%. On the other hand, the query performance result patterns in Figure 8.14 (b) show an exponential growth along with the growth in data sizes for the XPath query and a linear growth for the semantic XPath query. The semantic queries outperform the original XPath query by a percentage between 98% and 99.5%.

The significant query performance difference shown by the XPath queries and their semantic XPath queries indicate a substantial evaluation required when XPath axes and predicates are used in the XPath query.

In the case of XPath queries in Figure 8.14, the evaluation requires both downward and horizontal directions due to the use of the **preceding-sibling** axes. In addition, it appears that the lack of support for processing XPath axes for certain types of XML-enabled databases also contributes to the slow performance. It so happens that the database chosen for this evaluation faced this challenge. Of course, not all XML-enabled databases would follow this trend. However, this evaluation result leads us to believe that our semantic transformations are very useful in preparing similar XML-enabled database systems to meet the challenge of processing such query XPath axes.

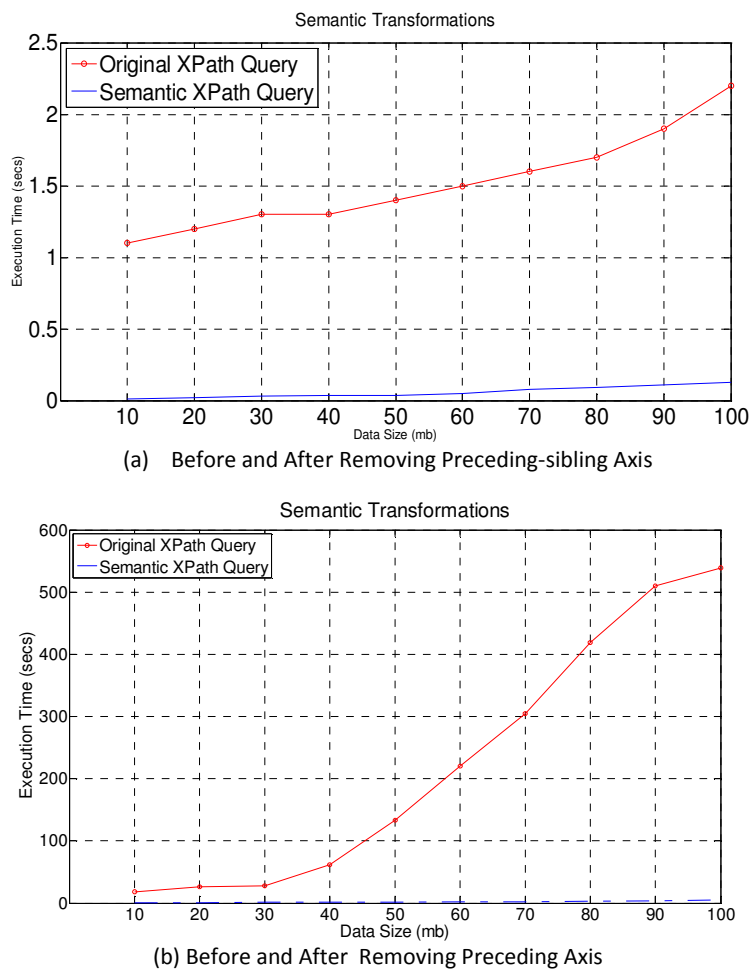


Figure 8.14 Semantic Transformation of XPath Queries with Combination of Axes

8.5 Semantic Conflict Detection

One of the benefits of semantic transformation is that a query can be answered without any need to access the database. Often the users are unaware of the structure

of the data source. The users may issue XPath queries that do not satisfy the semantics. When such situations are encountered, an XPath query will definitely return an empty result. With semantic transformations, such XPath queries can be detected avoiding the waste of unnecessary resources. Such detection is referred to as *semantic conflict detection*.

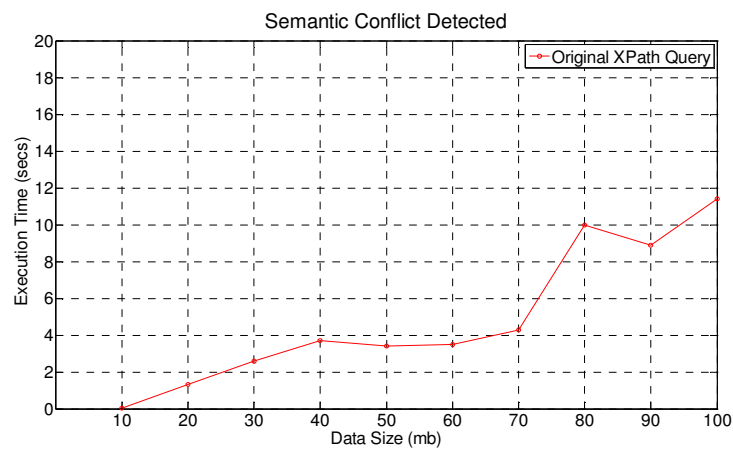
Semantic conflict detection works like a satisfiability study of XPath queries [Benedikt et al., 2005; Figueira, 2009; Ishihara et al., 2010]. The unsatisfied query problem is linked more with the complexity of XPath queries, both syntactically and semantically. However, due to the earlier use of satisfiability of semantics defined in DTD, this work uses semantics in XML schemas to complement the earlier work on satisfiability.

Prior to the query transformation, the semantic transformation will ensure that there are no conflicts such as element name, query structure and irregularities in the XPath expression. This can be achieved by using the semantics derived in given unique paths and the constraints of elements. This section demonstrates a number of XPath queries that are specified incorrectly in terms of either data structures or semantics. It also shows the query performance of XPath queries without the transformation. When these queries are transformed, the semantic conflict detection is triggered, which will inform the user that no data is returned for a particular XPath query.

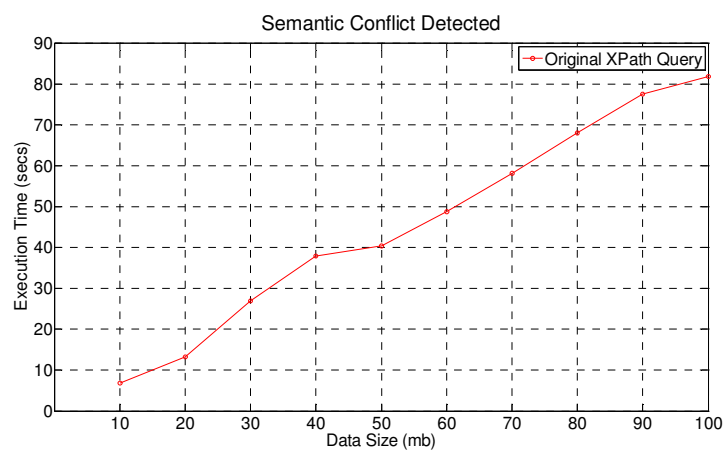
Figure	XPath Query	Conflicts
8.16a	<code>dblp/article[year <1950]/title/tn</code>	Semantic conflict is detected in condition as comparison value is out of range value defined in the Schema.
8.16b	<code>dblp/phdthesis[supervisor = /dblp/book/@id]/title/tn</code>	Semantic conflict is detected in condition where path comparison value /dblp/book/@id is not valid.
8.16c	<code>//book/title[2]/tn</code>	Semantic conflict is detected in index position condition 2 which is not within occurrence range.
8.17a	<code>dblp/article/title/following-sibling::author</code>	Semantic conflict is detected in structural element order as author is preceding sibling of title in the XML Schema.
8.17b	<code>dblp/article/title/preceding::book</code>	Semantic conflict is detected in structural element order as book follows article in the XML Schema.

Figure 8.15 XPath Queries with No Semantic XPath Queries

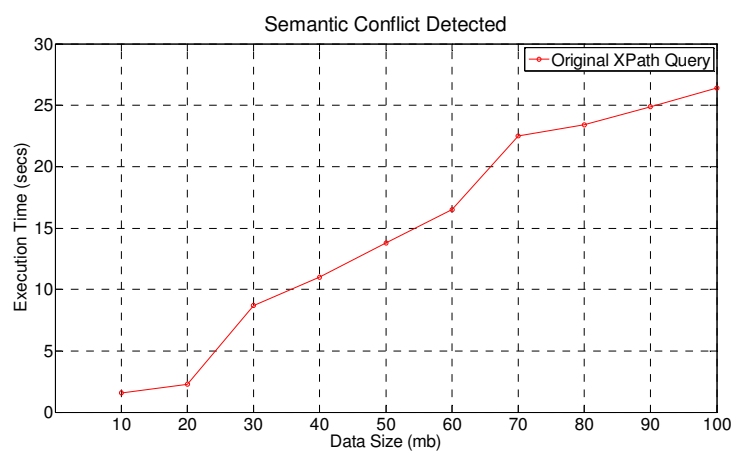
Figure 8.15 demonstrates a series of XPath queries that do not have semantic counterparts due to the conflicts detected in them.



(a)



(b)



(c)

Figure 8.16 Before and After Applied Semantic Conflict Detection of XPath Transformation

The XPath queries in Figure 8.16 demonstrate the semantic conflicts detected in the condition on the element that definitely returns an empty result.. The query performance result patterns in Figure 8.16 (a, b & c) indicate a linear increase along with an increase in data sizes regarding the empty results. This indicates that there is a high consumption of resources during the search for answers for those queries.

The XPath queries in Figure 8.17 demonstrate the semantic conflicts detected in path elements with regards to the order constraint of the element that definitely returns an empty result.

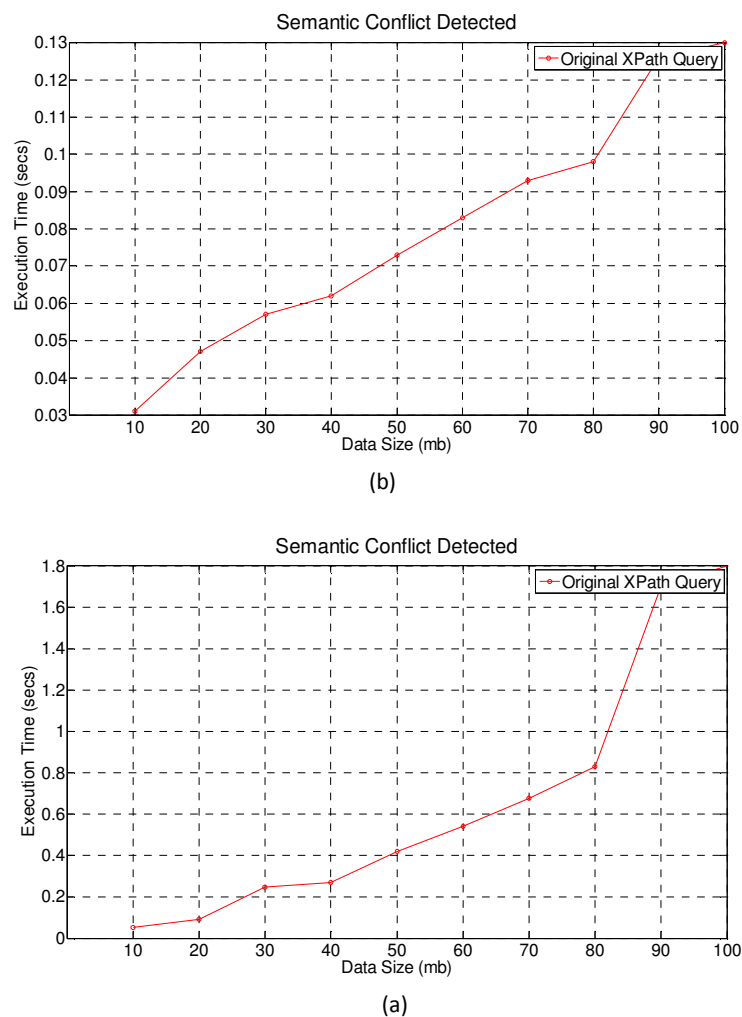


Figure 8.17 Before and After Applied Semantic Conflict Detection Transformation

The patterns of performance result patterns in Figure 8.17 (a & b) indicate a linear increase along with an increase in data sizes. The resulting pattern demonstrates that there is a significant amount of execution time spent on the search for answers for those queries. By using semantic transformation, the average time taken to transform

the XPath queries shown in Figure 8.17 is between 0.091 and 0.186 seconds, which is negligible when compared to the time take by the original queries.

8.6 Summary

This chapter has presented an evaluation of the proposed techniques for semantic transformations using a real data set, that of DBLP together with the accompanying DBLP XML Schema and an off-shelf commercial XML-enabled database management system. The evaluation is carried out using incrementally scaled data sets from the DBLP data set, which allows a thorough analysis of the impact of query performance. For each main category of semantic transformations, a set of XPath queries is specified and then by applying specific semantic transformation within each category, one or more semantic XPath queries are produced. The semantic XPath queries and their original XPath queries produce equivalent result sets. The experimental evaluation enables us to determine the semantic transformation as an optimization device summarized in Figure 8.18.

Semantic Transformation	Optimization Devices	Optimization Devices with Limitations
Semantic Path Expansion	✓	As long as '/' is expanded to one single path fragment, this certainly reduces the path evaluation and ultimately query processing.
Semantic Path Complement	✓	As long as '.' and the condition element are removed from the XPath query, this certainly reduces the path evaluation and ultimately query processing.
Semantic Path Contract		✓ Optimized when '*' is replaced with '/' and * represented multiple paths fragments or elements
Predicate Elimination		✓ Optimized when <ul style="list-style-type: none"> • Comparison elements are made up by a path fragment, with or without a comparison value • Comparison value is an absolute path
Predicate Reduction	✓	✓ Optimize when <ul style="list-style-type: none"> • Comparison elements are made up by a path fragment, with or without a comparison value • Comparison value is an absolute path
Preceding- or Following-sibling		Significantly optimize
Following or Preceding	✓	Significantly optimize.
Ancestor / Ancestor-or-self	✓	Confidently optimize when ancestor-or-self axis is removed.
Parent	✓	Significantly optimize
Descendant or descendant-or-self		Optimize when '*' is replaced with '/'

Figure 8.18 Identifying Semantic Transformations as Optimization Devices

Chapter 9

Experimental Evaluation Using Benchmark Data Sets

This chapter focuses on the effectiveness of the proposed semantic transformations by conducting experimentation on Michigan benchmarking data sets. Most of the experimental queries are the Michigan queries. However,

As discussed in Chapter 3, XML data is a tree structure expression where the depth of the data hierarchy can greatly influence the performance of query processing. This research has found that most of the real application data sets have shallow data hierarchies that make the query structures less expressive. This chapter focuses on an evaluation of expressiveness of data hierarchies by exploring more complex query structures and predicates in XPath queries.

9.1 Performance Evaluation

As described in Chapter 7 in this experimental design, each XPath query and its semantic XPath query over each data set would be executed n runs. The execution time is accumulated for the last three runs. The average execution time is then produced based on the last three runs. The average execution time is referred to as

the *performance result* throughout this chapter. While the performance result for an original XPath query is the average execution time, the performance result for its semantic XPath query is the average execution time plus the transformation time. The evaluation involves the comparison between the performance results of the original XPath query and its semantic XPath query.

The reader is reminded that the constraints used to transform XPath queries are the information in lists Q and C , which have been proposed in Chapter 4. While Q contains a list of possible unique paths (full path expressions) derived from XML schema, list C contains a list of constraints of elements defined in XML schema.

From the available Michigan XPath queries, the evaluation of XPath queries is divided into three categories: *Single Query condition of Value-based Comparison*, *Joined Query conditions with Value-based Comparison* and *Twig Join with Value-based Comparison*. This experiment focuses on the XPath queries that can be executed using the adopted XML-Enabled database management system. In short, the performance of XPath queries specified with axes is not central to the study in this chapter.

An XPath query and its semantic XPath query/queries are equivalent if and only if they produce the same result set although they may be different in structure.

9.1.1 Single Query Condition with Value-based Comparison

This section presents the performance results of a set of XPath queries specified with predicates that have only a single query condition. Figure 9.1 summarizes the information related to XPath queries used by the experiment in this section that includes the Query Type, XPath query, Semantic XPath query that is produced after being transformed, and the Result Type.

Figure 9.2 shows the performance results of the benchmark XPath query `//eNest[@aLevel=2]` and the equivalent semantic XPath query. For this XPath query, *semantic path expansion* and *predicate elimination semantic transformations* are applied to obtain the equivalent semantic XPath query.

Figure	Query Type	XPath	Semantic		Selection Return
			Path Expansion	Predicate Elimination	
9.2	Ancestor-Descendant	//eNest[@aLevel=2]	/eNest/eNest[@Level=2]	/eNest/eNest	Sub-tree
9.3		//eNest[@aLevel=7]/ eNest[position()=2]/ @aUnique1	/eNest/eNest/eNest/eNest/ eNest/eNest/eNest[@Level =7]/eNest[position()=2]/@a Unique1	/eNest/eNest/eNest/eN est/eNest/eNest/eNest/ eNest[position()=2]/@a Unique1	Values of field
9.4		//eNest[@aLevel=16]	/eNest/eNest/eNest/eNest/ eNest/eNest/eNest/eNest/e Nest/eNest/eNest/eNest/eN est/eNest/eNest/eNest [@Level=16]	/eNest/eNest/eNest/eN est/eNest/eNest/eNest/ eNest/eNest/eNest/eNe st/eNest/eNest/eNest/e Nest/eNest	Sub-tree

Figure 9.1 XPath Queries and Semantic XPath Queries

The semantic path expansion transformation first expands `//eNest[@aLevel=2]` to `eNest/eNest[@aLevel=2]` based on the unique path `eNest/eNetst` located in list Q . The first semantic XPath query is a full path that represents the number of data hierarchies based on the value of the query condition, which in this case is 2.

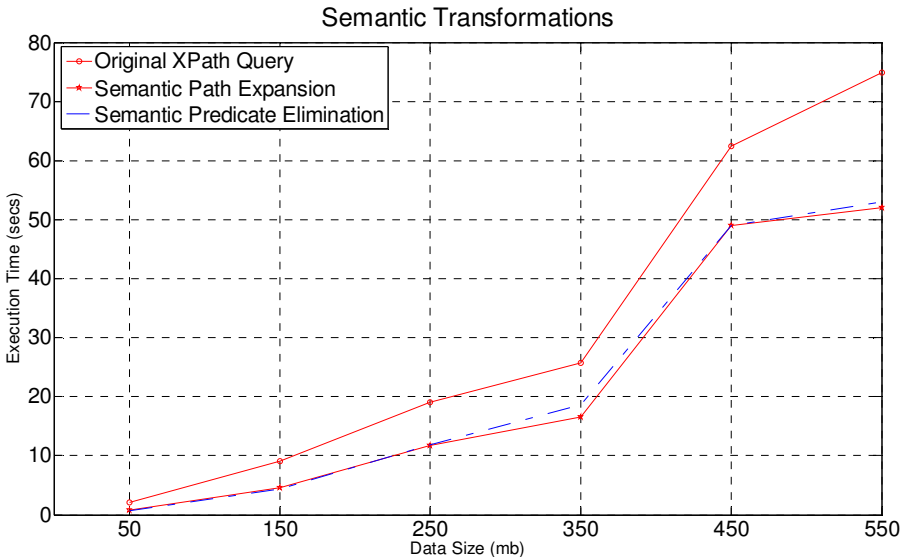


Figure 9.2 Performance Results Before and After Semantic Transformations

The second semantic transformation removes the predicate [**aLevel=2**] from the semantic XPath query `/eNest/eNest[@aLevel=2]` by applying predicate elimination semantic transformation. In this way, we study the impact of query performance without using the predicate which is considered as a redundant component after the XPath query has been expanded to a full path expression. Note that the semantic path

expansion must be applied before removing the predicate. That is because the predicate indicates the levels of data hierarchies in the XPath query.

The performance results in Figure 9.2 show a linear increase along with the increase in data sizes. As expected, among the three patterns of results, the query performance of the original XPath query is the worst and the performance results of two semantic XPath queries are almost the same. The reduction of predicate does not reduce much of the performance as expected, as it actually reduces the verification of attributes; however, the result does not reflect what is expected. The negligible result of removing the predicate is possibly caused by the shallowness of the data hierarchy indicated by the value of 2 in the query condition. To verify this, another similar query with further data hierarchies based on the value of the query condition is evaluated next.

Figure 9.2 show the performance of both semantic XPath queries (the former is marked with Semantic Path Expansion and the latter is applied with Semantic Predicate Elimination transformations) outperforms the original XPath query by between 30% and 50%. There is a linear increase of performance results along with the increase in the size of the data sets. Hence, it can be expected that the improvement produced by the semantic XPath queries will continue to increase along with the growth in the larger data sets.

The next XPath query used for evaluation is to increase the data hierarchy by increasing the value of the query condition. Figure 9.3 provides further evidence for the argument previously made about the cause of the negligible performance result after the predicate has been removed as in Figure 9.2. The performance results in Figure 9.3 are for the XPath query `//eNest[@aLevel=7]/eNest[position()=2]/@aUnique1` and its equivalent semantic XPath queries.

As the query condition has value 7, the semantic path expansion replaces `//eNest` with `/eNest/eNest/eNest/eNest/eNest/eNest/eNest` which is a unique path located in Q . The first semantic XPath query `/eNest/eNest/eNest/eNest/eNest/eNest/eNest[@aLevel=7]` is now produced

showing a full path that has 7 data hierarchies based on the value **7** of the query condition.

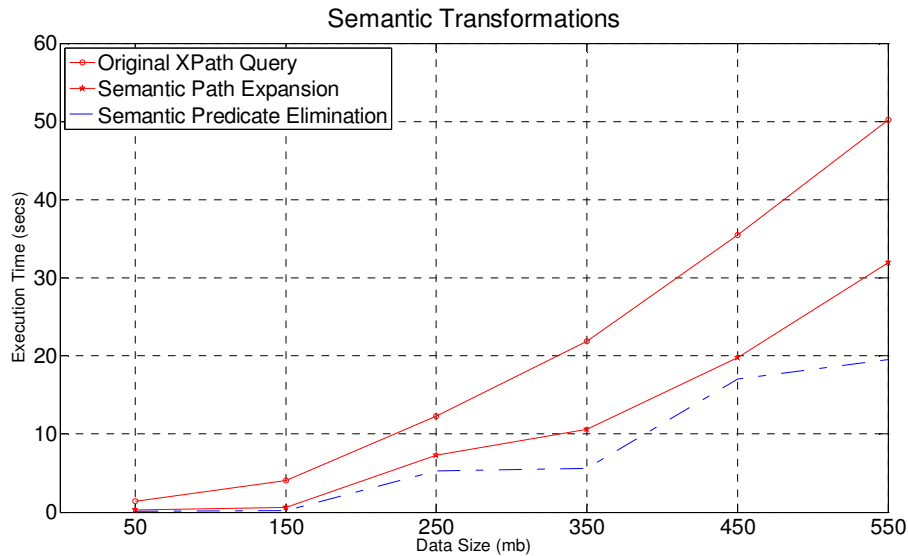


Figure 9.3 Performance Results Before and After Semantic Transformations

Since the XPath query has already been expanded to a full path, it is not necessary to use the predicate that has a query condition to restrict the data hierarchy of the full path. The semantic transformation applies a predicate elimination semantic transformation to remove the predicate `[@aLevel=7]` from the first semantic XPath query. The query performance result patterns for XPath query and semantic path expansion XPath query in Figure 9.3 grow exponentially along with the growth in data sizes. The query performance result pattern of the predicate reduction XPath query increases linearly along with the data sizes. In summary, the semantic path expansion XPath queries outperform the original XPath query by 30% to 70%.

The semantic XPath query, which is obtained by predicate elimination semantic transformation, outperforms the semantic path expansion XPath query by between 30% and 50%, and outperforms the original XPath query by 50% to 70%. This means that the semantic XPath query rewriting from the predicate elimination semantic transformation performs the best. This supports the conclusion that when the XPath query is expressive in structure, the semantic path expansion significantly improves the query performance. Following the application of semantic path expansion transformation and the removal of predicates, the performance improvement is even more promising.

Based on the performance result shown in Figure 9.3, especially the query performance result after the removal of the predicates, it can be concluded that the data hierarchy used in the XPath query condition greatly contributes to query performance. It has been proven that the deeper the data hierarchy used by the query condition, the greater will be its effect will be on the query performance despite the full expansion of the path as in Figure 9.3.

Predicate elimination semantic transformations undoubtedly benefit XPath queries that delve deeply into data hierarchies. Some data tree structures are fan-out structures [Runapongsa et al., 2006] meaning that the deeper the tree grows, the greater its number of branches, also known as a fan-out. This means that when the query delves into a deeper data hierarchy, there is a possibility that more data will be returned and searching of the data tree is significantly increased. If the verification of redundant components during processing can be eliminated, as was done in the XPath query above, there is a significant gain in performance.

Next, another evaluation is performed based on the depth of data hierarchy. The lowest hierarchy in the data tree is used for this experiment.

Figure 9.4 shows the performance results of the XPath query `//eNest[@aLevel=16]` and its equivalent semantic XPath query.

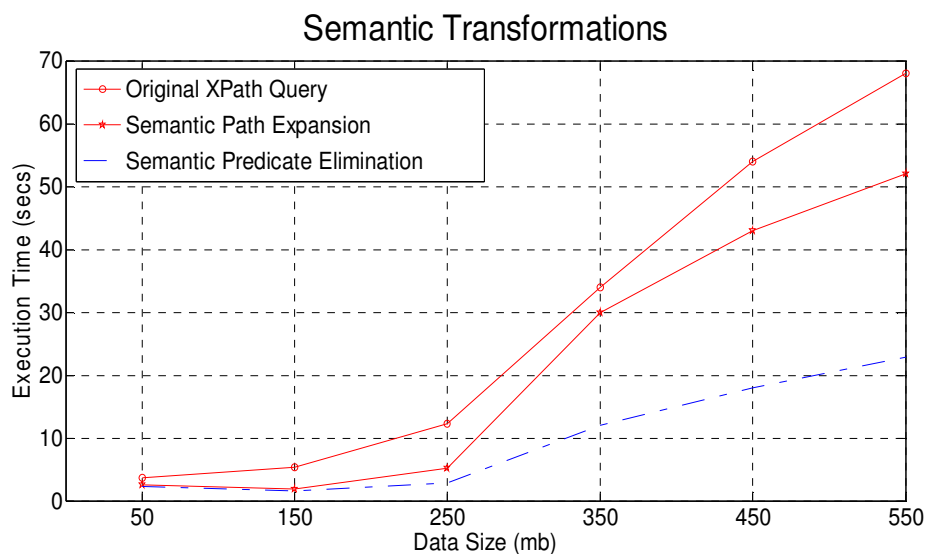


Figure 9.4 Performance Results Before and After Semantic Transformations

The semantic path expansion transformation first expands `//eNest[@aLevel=16]` to a full path that has 16 hierarchies, which is restricted to the value of 16 in the query condition. This full path is the located unique path in Q . The first semantic XPath query is the full path together with the predicate `[@aLevel=16]`. Following the first semantic transformation, the predicate elimination semantic transformation eliminates the predicate `[@aLevel=16]` from the first semantic XPath query.

The XPath query `//eNest[@aLevel=16]` is similar to the earlier XPath queries in Figures 9.2 and 9.3, except that the hierarchy of the XPath query in Figure 9.4 is much deeper; in fact it traverses the lowest data hierarchies in the Michigan XML document tree.

The graphs in Figure 9.4 indicate the performance results obtained by the XPath query and its semantic XPath queries grow linearly as the data sizes increase. The performance result obtained by applying semantic XPath query semantic path expansion transformation is better than the performance result obtained by the original XPath query by between 25% and 40%. However, the most significant performance improvement, between 50% and 70%, is achieved by the semantic XPath query in which the predicate is eliminated.

The query performance results in Figure 9.4 demonstrate that the deeper the hierarchy data, the slower the XPath query will perform. The transformation of such an XPath query to a full path expression and the reduction of query conditions, if possible, would boost performance significantly.

The performance results obtained by XPath queries and their semantic XPath queries confirm the benefits of semantic path transformation and predicate elimination semantic transformations. Those semantic transformations would certainly provide performance improvement for similar data hierarchies and XPath query structures.

9.1.1 Multiple Query Condition with Value-based Comparison

This section presents the query performance results of a set of XPath queries specified with predicates that have query conditions joined with operators AND or OR or both.

Figure 9.5 summarizes the information related to XPath queries used in the experiment in this section, including Query Type, XPath queries, Semantic XPath that is produced after being transformed, and the Result Type.

Figure	Query Type	XPath	Semantic		Selection Return	
			Path Expansion	Predication Reduction		
9.6	Ancestor-Descendant, Conjunctive & Disjunctive Conditions	//eNest[@aLevel = 17 or @aLevel =14 and @aFour=1]	/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest[eNest/@aLevel = 17 or @aFour=1]	//eNest [@aLevel =14 and @aFour=1]	Sub-tree	
9.7		//eNest[eOccasional /@aRef and @aSixtyFour =0]	-----	//eNest[eOccasional/@aRef] (1)		
				//eNest[@aSixtyFour=0] (2)		
9.8		Ancestor-Descendant, Disjunctive Condition	eNest[@aLevel =12 and eOccasional/@aRef and @aSixtyFour =0]	/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest[eNest/@aLevel = 12 and eOccasional/@aRef and @aSixtyFour =0]		/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest[eOccasional/@aRef] (1)
						/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest[eOccasional/@aSixtyFour=0] (2)

Figure 9.5 XPath Queries and Semantic XPath Queries

Figure 9.6 presents the performance of the XPath query `//eNest[@aLevel =17 or @aLevel =14 and @aFour=1]` and its equivalent semantic XPath queries. Note that the query condition `@aLevel =17` is not specified by the actual benchmark query, it has been added for a special purpose to perform a mutual exclusion described in detail below.

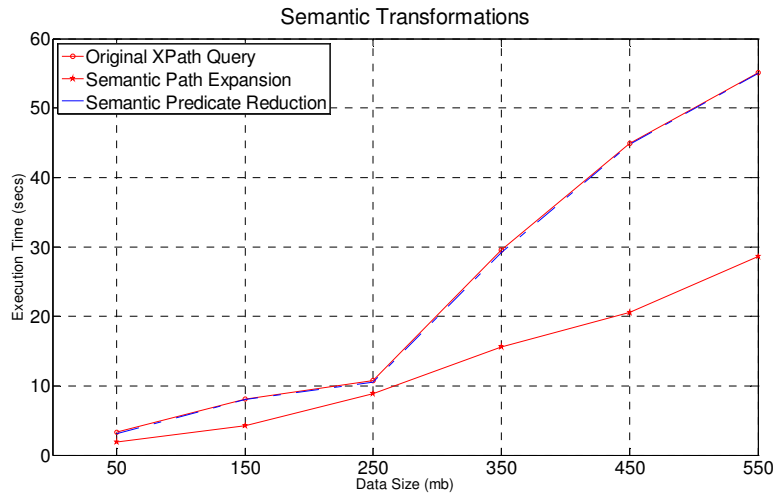


Figure 9.6 Performance Results Before and After Semantic Transformations

As stated in Chapter 7, the Michigan benchmark data has only 16 hierarchies; this means **@aLevel =17** is regarded as a conflict attribute. When a conflict query condition exists and it is joined by OR with another query condition, it is known as a mutual exclusion (refer to Chapter 3). To demonstrate the mutual exclusion, **@aLevel =17** is added to show how it allows the co-existence of two query conditions **@aLevel =14** and **@aFour=1**. The existence of the query condition **@aLevel =17** does not affect the overall result.

The first step in the semantic transformation applies the predicate reduction semantic transformation to remove the query condition **@aLevel =17** from the original XPath query. This produces the first semantic XPath query.

Following the first semantic transformation, the second semantic transformation is to apply the semantic path expansion transformation that removes the query condition **@aLevel=14** from the predicate based on a matched unique path located in *Q*. The matched unique path contains 14 data hierarchies of **eNest**.

Notice that in this semantic path expansion transformation, the valid hierarchy query condition **@aLevel =14** is removed but it is joined by AND to the next query condition; this is allowed as long as the full path represents the expected data hierarchies restricted by the query condition **@aLevel =14**. This transformation is done because the query condition **@aLevel=14** refers to a very deep hierarchy data. The simultaneous removal of a hierarchy query condition and expansion of the XPath expression would reduce the validation of each data hierarchy every time it is

accessed. At the same time, it also guarantees a significant improvement in query performance. This has been proven in XPath queries in Figures 9.3 and 9.4.

The graph in Figure 9.6 indicates the performance results obtained by the original and the equivalent semantic XPath queries increasing linearly, along with the increase in data sizes. The semantic path expansion XPath query outperforms the original XPath query by an average of 98%. However, there is no difference in performance results between the original XPath query and the semantic XPath query produced by applying the predicate reduction semantic transformation to the original XPath query. This shows that the removal of the mutual exclusive query condition is not significant. Recall that the performance result of this semantic XPath query is the average execution time plus the transformation time. Therefore, based on the result depicted by the graph in Figure 9.6, there is only a slight improvement in performance.

The scope of this research does not include a study of the storing and mapping techniques of the database engine, especially for those XML-Enabled Databases. Different XML-Enabled Databases may have different techniques to achieve these kinds of storage objectives. To be able to comment on this, a broader investigation of several systems is required and will be considered as a future extension of this research.

Up to this point, this chapter has demonstrated the semantic transformations using a set of XPath queries that are specified with query conditions, conditions that are joined based on data hierarchies using an OR or AND operator. The query conditions are also based on the attributes that appear to be on the same data hierarchy. For example, the XPath query in Figure 9.6 has a query condition **@aLevel =14** join with the query condition **@aFour=1**. For every **eNest** at hierarchy 14th, the attribute **aFour** exists.

The hierarchy query condition is eliminated and replaced by the full hierarchical path expression that speeds up the traversing direction. This has been achieved by the previous XPath queries in Figures 9.2 to 9.4 & 9.6. On the other hand, removing query conditions from the XPath queries predicate does not always produce benefits,

especially when a query condition is non-hierarchical and restricts the searching space. We show this in the XPath query of Figure 9.7.

In Figure 9.7, we demonstrate the performance results of the XPath query `//eNest[eOccasional/@aRef and @aSixtyFour =0]` and its equivalent semantic XPath query. The XPath query has a predicate containing a set of query conditions based on the attributes other than the hierarchy attribute. As can be seen, there is no query condition that indicates the hierarchy of the data element; therefore, it is expected that the searching sometimes covers the whole document.

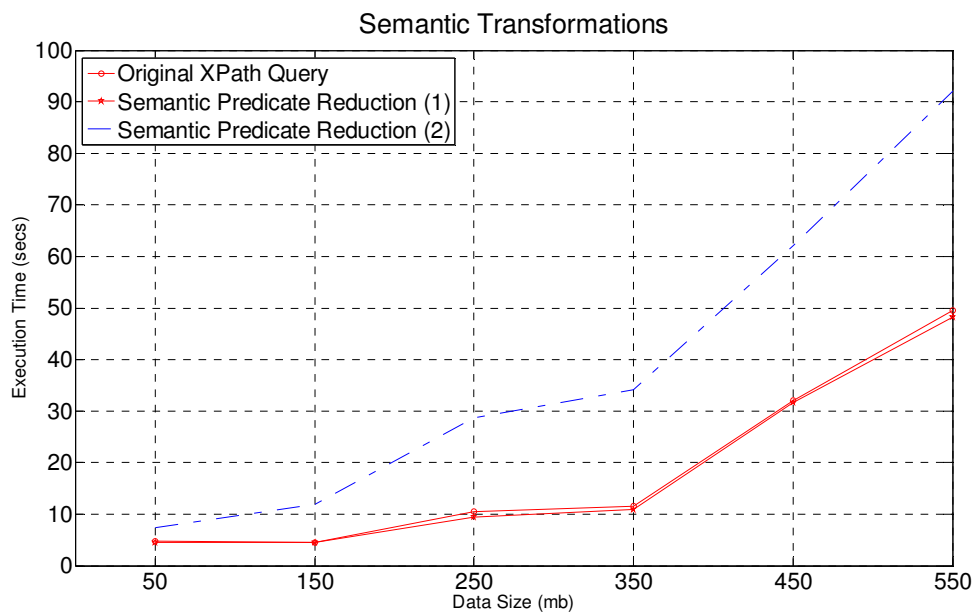


Figure 9.7 Performance Results Before and After Semantic Transformations

The first semantic transformation removes the query condition `@aSixtyFour = 0` in accordance with the constraint of **eOccasional** existence in list *C*, which is a dependant of **aSixtyFour** that have value of 0. However, not all **eNest** has **eOccasional**. This means that as long as the **eOccasional** element is present, `@aSixtyFour` expects to have a value of 0. This does not work in reverse, which leads to the second semantic transformation that applies predicate reduction semantic transformation to the original XPath query. This time the query condition `eOccasional/@aRef` is removed instead of the query condition `@aSixtyFour = 0`. There is a significant difference when one but not the other is removed.

The first predicate reduction semantic transformation produces the first semantic XPath query **//eNest[eOccasional/@aRef]**. The second predicate reduction semantic transformation produces the second semantic XPath query **//eNest[@SixtyFour = 0]**.

The performance result patterns in Figure 9.7 increase exponentially along with the increase in data sizes. As can be seen, there is a significant improvement of almost 100% in query performance obtained by **//eNest[eOccasional/@aRef]**, which is the first semantic XPath query.

In the second semantic transformation, the query condition **eOccasional/@aRef** is removed from the original XPath query in accordance with the constraints set for **aSixtyFour = 0** in the schema that guarantees the existence of **eOccasional**. The semantic XPath query **//eNest[@SixtyFour=0]** is produced by applying predicate elimination semantic transformation. The semantic XPath query shows no difference in performance compared with the performance result obtained by the original XPath query. This is because the data of **eNest** elements do not have to be accessed if the attribute **SixtyFour** of **eNest** is not 0.

Sometimes, additional query conditions are introduced to the predicate enabling the query to perform more efficiently. This transformation is known as *semantic introduction* [Chakravarthy et al., 1986a]. The semantic introduction transformation has been introduced to optimize queries in deductive databases. Currently, this research does not consider *semantic introduction* in XML databases because such a transformation requires reasoning theory in order to accomplish the task, which is beyond the scope of this research. This research uses the standard W3C semantics in XML Schemas. This kind of transformation will be considered for a future work.

Figure 9.8 shows the query performance results of the XPath query **//eNest[@aLevel =12 and eOccasional/@aRef and @aSixtyFour =0]** and the equivalent semantic XPath queries. In this XPath query, a new query condition **@aLevel =12** is added to the predicate. The new query condition allows the query hierarchy to be expanded to its full path and it also allows a smaller amount of data to be returned.

When applying the semantic path expansion transformation, **@aLevel =12** is removed and **//eNest** is expanded to a full path of 12 hierarchies according to an

existing unique path located in list Q . The application of an XPath query semantic path expansion transformation shows a significant linearly increasing pattern of performance improvement of between 50% and 80% along with the increase in data sizes.

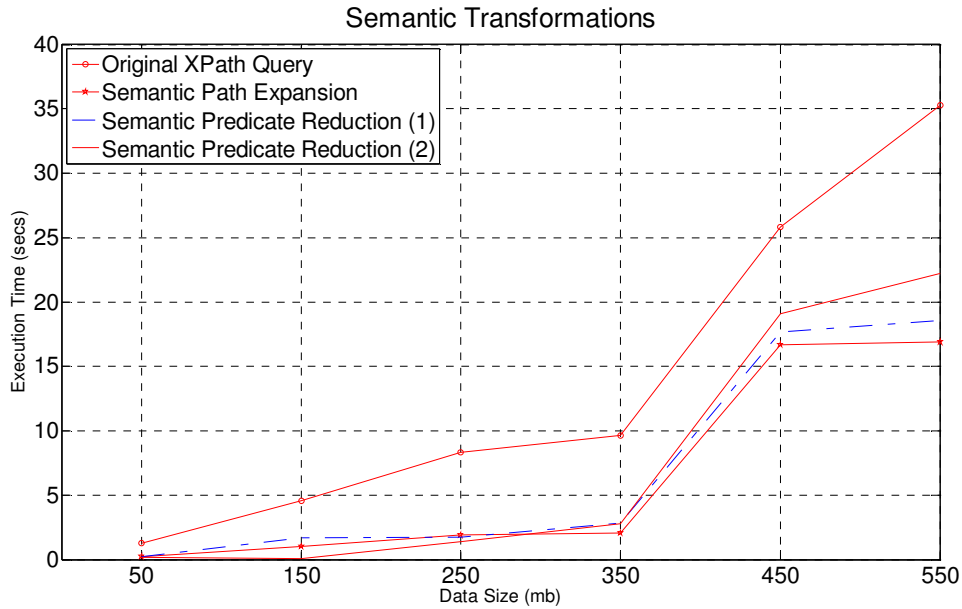


Figure 9.8 Performance Results Before and After Semantic Transformations

Next, the predicate reduction semantic transformation is applied to remove **eOccasional/@aRef** from the first semantic XPath query as when **@aSixtyFour = 0**, it guarantees the existence of **eOccasional/@aRef**. The second semantic XPath query is a full path expanded to 12 hierarchies with a predicate [**@aSixtyFour = 0**]. As indicated earlier, Figure 9.7 shows how **eOccasional/@aRef** and **@aSixtyFour** work with each other. Therefore, the third transformation applies the predicate reduction transformation to remove [**@aSixtyFour = 0**] from the first semantic XPath query. The third semantic XPath query is a full path expanded to 12 hierarchies with a predicate [**eOccasional/@aRef**]

Figure 9.8 shows that the performance result in it increases linearly along with the growth of the data sizes. The semantic XPath query, with semantic path expansion transformation applied, outperforms the original XPath query by between 50% and 80%. The second semantic XPath query (where predicate reduction semantic transformation is applied) shows a query performance improvement of between 10% and 30% along with the increase in data sizes.

As can be seen, the removal of query conditions **eOccasional/@aRef** or **@aSixtyFour = 0** from the semantic path expansion XPath query, have not contributed to an improvement of query performance. It is suspected that a special means of structuring and storing data in this particular XML-Enabled database was required, of which users are not aware. For this reason, in the near future, there should be a more thorough investigation into this kind of query patterns using a similar data structure.

9.1.2 Twig Pattern Query Conditions with Value-based Comparison

So far, XPath queries that are specified with a single predicate containing simple query conditions, have been demonstrated. In this section, XPath queries specified with multiple predicates that represent simple or twig pattern query conditions are used for further experiments. These complex query conditions, carried out over real data sets, as in the previous chapter, are challenging due to the shallow depth of the data structure; the benchmark data sets in this chapter address this challenge.

Figure	Query Type	Original XPath	Semantic		Selection Return
			Path Expansion	Axis Elimination	
8.11	Ancestor-Descendant, Parent-child Condition and Self Axis	//eNest[@aLevel = "11"] [./eNest/@aFour="3"]/@aUnique1	/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest[eNest[./eNest/@aFour="3"]/@aUnique1]	/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest[eNest/@aFour="3"]/@aUnique1	Values of Attribute
8.12	Twig Pattern, (Parent-child and	//eNest[@aLevel="8"] [./eNest[@aSixtyFour="3"]] [./eNest[@aSixtyFour="3"]]	/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest[eNest[./eNest[@aFour="3"]] [./eNest[@aSixtyFour="3"]]]	/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest[eNest[./eNest[@aFour="3"]] [eNest[@aSixtyFour="3"]]]	Sub-tree
8.13		//eNest[@aLevel="12"] [./eNest[@aFour="3"]] [./eNest[@aSixtyFour="3"]]	/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest/eNest[eNest[./eNest[@aFour="3"]] [./eNest[@aSixtyFour="3"]]]	/eNest/eNest/eNest/eNest/eNest/eNest/eNest[eNest[eNest[@aFour="3"]] [eNest/[@aSixtyFour="3"]]]	

Figure 9.9 XPath Queries and Semantic XPath Queries

Figure 9.9 shows a list of XPath queries and information pertaining to the transformations such as Query Type, Original XPath queries, Semantic Transformation and Result Type.

Figure 9.10 shows the performance results of the XPath query `//eNest[@aLevel = "11"][/eNest/@aFour="3"]/@aUnique1` and the equivalent semantic XPath queries.

The XPath query demonstrates the use of multiple predicates and *self* axis, optionally specified with operator “.”. This is not a twig query because the first query condition projects on an element.

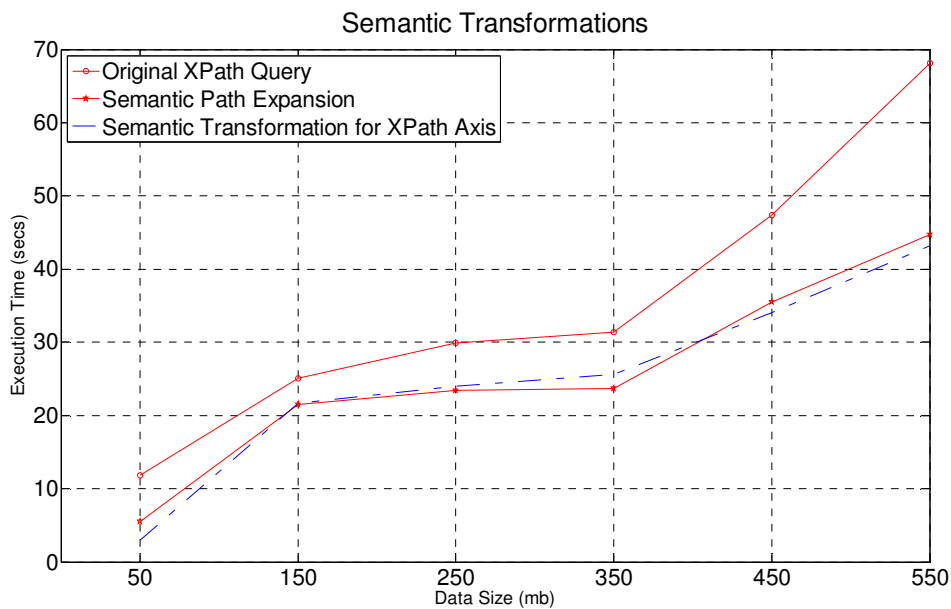


Figure 9.10 Performance Results Before and After Semantic Transformations Applied

The first semantic XPath query is produced by applying double semantic transformations. The first is semantic path expansion transformation to expand `//eNest` to a full path that has eleven elements `eNest` using only the parent-child ‘/’ relationships. This is due to the unique path located in Q that satisfies the query condition value of 11. The second is to remove the first predicate `[@aLevel = "11"]`. The second semantic XPath query is produced by applying semantic transformation to the *self* axis to remove the operator ‘.’ from the second predicate and the third predicate.

The performance result patterns in Figure 9.10 show a linear increase along with the growth in data sizes. The semantic XPath query obtained by the semantic path expansion outperforms the original XPath query by between 40% and 50%. However, the equivalent semantic XPath query without the operator “.” axis does not seem to perform better than the first semantic XPath query. Nevertheless, it is not the worst because its fluctuating pattern is not much different from the result obtained by the first semantic XPath query. For this reason, it can be concluded that both semantic XPath queries perform at the same rate of improvement and they both outperform the original XPath query.

As can be seen from the semantic XPath queries, when the XPath query operator ‘//’ is used, the impact is significant. However, this may not be the only factor that affects the performance, because as explained earlier, this impact can also be caused by the structure of the individual database engine which is not within the scope of this research. However, if the impact is caused by some implicit ways of specifying queries, this can be handled. In this case, path component such as ‘//’ has been eliminated by the semantic path expansion transformation. The result shows a great improvement in performance after this elimination.

The next evaluation demonstrates the use of more complex query conditions in an XPath query, which is referred to as a ‘twig pattern’ query condition.

Figure 9.11 shows the performance result obtained by the XPath query `//eNest[@aLevel="8"][/eNest[@aFour="3"]]/eNest[@aSixtyFour="3"]`. Due to the nested query condition in the second and the third predicates `[/eNest[@aFour="3"]]` and `[/eNest[@aSixtyFour="3"]]` respectively, this XPath query is known as the twig pattern XPath query type [Runapongsa et al., 2006]. Such a query condition in the second predicate is equivalent to a fragment of comparison paths `./eNest/@aFour` and `./eNest/@aSixtyFour` in which each has comparison values of 3.

In summary, the given XPath query has three predicates. While the first predicate is based on a single element that restricts the hierarchy of the accessed data, the last two predicates are based on query conditions that make the XPath query a twig pattern type.

The equivalent semantic XPath query is produced by applying the semantic path expansion transformation which enables **//eNest** to be expanded to a full path of 8 **eNest** elements using only the parent-child relationship hierarchies. At the same time, it also applies predicate reduction semantic transformation to remove the predicate **[@aLevel="8"]** from the XPath query.

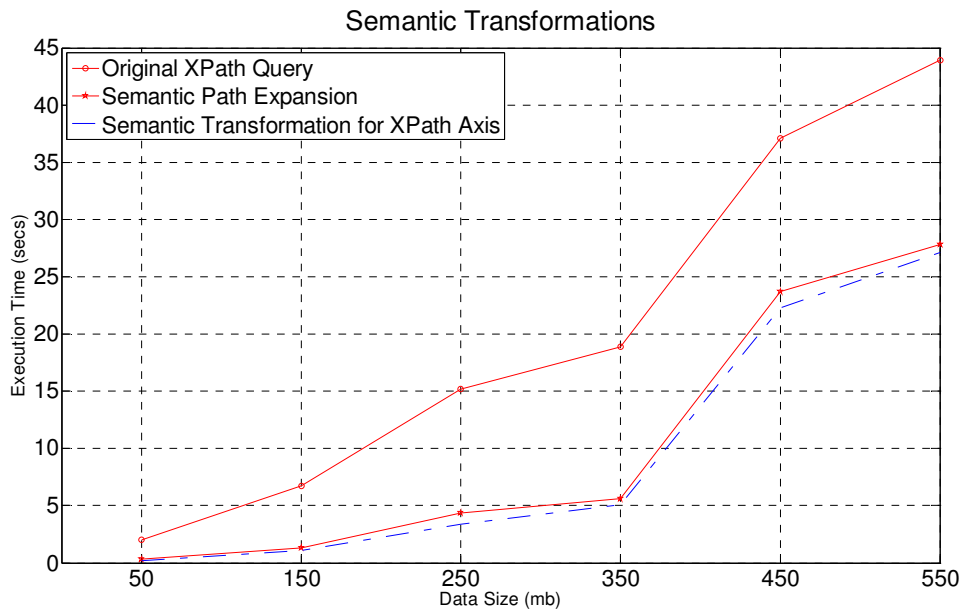


Figure 9.11 Performance Results Before and After Semantic Transformations Applied

The performance result obtained by the semantic XPath query, which is identified by the term **Semantic Path Expansion** in Figure 9.11, after applying semantic path expansion transformation, outperforms the performance result obtained by the original XPath query by between 25% and 70%. There is also a strong linear growth along with the increase in data sizes performed by both the XPath queries. This confirms that the larger the dataset, the greater the improvement is in query performance that can be achieved.

The semantic transformation of the XPath axis is applied on the first semantic XPath query to produce the second semantic XPath query, whose performance pattern is identified by the term **Semantic Transformation for the XPath Axis** in Figure 9.11. The semantic transformation removes the self “.” axis from the first semantic XPath query. The performance result in Figure 9.11 again indicates that the second

semantic XPath query outperforms the original XPath query by between 27% and 72%.

Even though there is a slight improvement in performance obtained by the second XPath query over the one performed by the first XPath query, both the semantic XPath queries perform almost the same. Therefore, it is better to remove unnecessary XPath components to optimize the performance.

Figure 9.12 shows the performance result of the selected twig pattern XPath query `//eNest[@aLevel="12"][/eNest[@aFour="3"]]/eNest[@aSixtyFour="3"]` and its semantic XPath queries.

The selected twig pattern XPath query in this case is very similar to the previous XPath query except that the predicate contains the query condition, which is used to restrict the selection of hierarchical data on a much deeper hierarchy of 12. As the second and third predicates are the same as those in the previous XPath query (Figure 9.11), it is not necessary to repeat the earlier analysis before transformation takes place.

The semantic XPath query is produced by applying double semantic transformations. The first semantic XPath query is produced by applying the semantic path expansion which enables the expansion of `//eNest` to a full path of 12 `eNest` that has only parent-child relationships. At the same time, predicate reduction semantic transformation is applied to remove the predicate `[@aLevel="12"]`.

The first semantic XPath query is then transformed once more by using semantic transformation for the axis where self ‘.’ axis is removed from the second and the third predicates. Hence, the second semantic XPath query is produced.

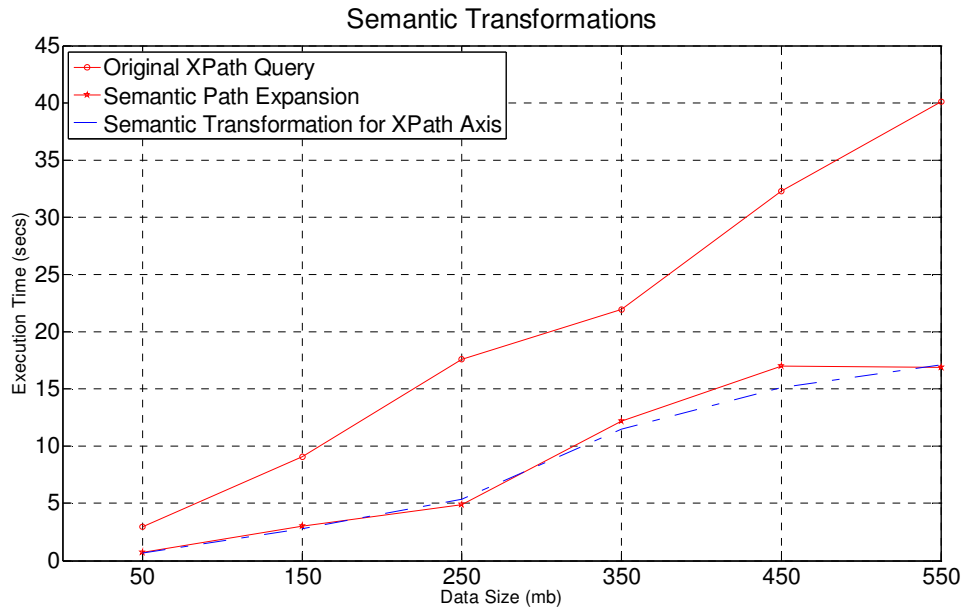


Figure 9.12 Performance Results Before and After Semantic Transformations Applied

The performance results in Figure 9.12 show a linear increase along with the growth in data sizes. There is a significant improvement of 100% for the semantic path expansion XPath query compared with the performance result obtained by the original XPath query. There is a slight decline in performance of the largest data set performed by the first semantic XPath query; this could probably be due to a hiccup in the processing memory.

The overall benefits of using semantic transformation are that both semantic XPath queries show a significant improvement compared with the performance of the original XPath query. Furthermore, according to performance results the performance improves with the increase in data sizes.

9.2 Summary

This chapter has conducted an evaluation of a series of selective benchmark XPath queries using benchmark incremental data sets. The evaluation is based on the semantic transformation techniques and XPath query components that are transformed in order to optimize the performance results.

The evaluation is divided into three sections:

- The first evaluation section focuses on a set of XPath queries, each of which has a predicate containing a query condition, which restricts the access to hierarchical data depending on the type of the hierarchy, that is whether it is shallow, medium or deep (the lowest hierarchy in the data tree).
- The second evaluation section focuses on a set of XPath queries, each of which has predicates containing simple query conditions that are joined by AND or OR. The query conditions restrict the access to hierarchical data based on data that are not necessarily hierarchical data.
- The third evaluation section focuses on a set of XPath queries, each of which has predicates containing twig patterns in a path fragment and is joined by AND.

Most benchmark XPath queries used in the evaluation show very promising improvement in performance after they have been transformed to equivalent XPath queries. In some semantic XPath queries, the performance result increases significantly after one or more query condition(s) have been removed. This is because the removal of a query condition plays a critical role in narrowing down the search space during the processing stage. However, there are cases when the removal of a query condition may affect the performance result, as shown in Figure 9.7. This discovery enables the scope of the research to extend to new findings in XML query transformations. The challenge of introducing extra query conditions to XPath queries is to use reasoning on semantics similar to what has been done using reasoning techniques for deductive databases.

Chapter 10

Conclusions and Future Work

The ever-increasing adoption of XML has created a need to ensure that XML query languages perform efficiently. Query optimization and transformation for XML query languages, both syntactically and semantically, have received a lot of attention by research communities in recent years. However, due to the fast progress of the application of XML data management solutions, XML-Enabled Database Management Systems still face several challenges. Among these challenges are query processing specific to query optimization. Semantic query optimization utilizes constraints in XML schemas to directly optimize a given query with a set of optimization rules. Due to the current complexity of the XML data structure, which is enabled by rich semantics in XML Schemas, semantic query transformations should be performed in a more systematic manner. This research has proposed a series of semantic transformations for XML queries for optimization purpose. In this chapter, a summary of the work of this thesis is presented, together with the main conclusion and an outline of topics for future work that may arise from this study.

10.1 Summary of the Contributions

Chapter 1 provided a broad overview of the XML background, XML database, XML schemas and query processing in XML. The description explored the following three concerns related to XML technology.

The first is the significant growth of XML data that leads to the development of XML schema in order to increase the supportability of further semantics to deal with data structure and the quality of XML documents.

The second is the critical concern of efficient storage management. XML document storage needs to be efficient. The management of permanent XML data must be able to deal with data independence, integration, access rights, versions, views, integrity, redundancy, consistency, recovery, and the enforcement of standards.

The third is a need to ensure that XML query languages perform efficiently. Therefore query optimization and transformation for XML query languages, both syntactic and semantic, have received a great deal of attention from research communities in recent years. Due to the rapid development of XML, query optimization still needs much attention.

The chapter describes the challenges of using semantics in XML Schema for optimization purposes for XML queries. This chapter explained the current complexity of the XML data structure which is enabled by the rich semantics in XML Schema. The utilization of semantics for query optimization purposes should be done in a more systematic manner. That can be done by leveraging the semantics from XML Schema and using it to investigate a set of semantic transformations.

The introductory section in Chapter 1 presents an overview of XML technology, XML schemes and XML Databases, and identifies the need for semantic transformations. Furthermore, it proposes the use of semantics in XML Schemas to overcome the limitations of semantics available in DTD for proposing semantic query transformations.

The chapter also provides the research motivation for semantic query transformations. That is, the process of applying semantics to optimize queries has been adopted by other databases including relational, object-oriented and deductive databases, and is also useful in XML databases due to the availability of XML Schemas. The research motivation clarifies the difference between semantic query transformations in XML database and other databases due to the types of XML

constraints in XML schemas which are driven by two sets of constraints: structural constraints and constraints imposed by elements.

Chapter 2 presented a survey and evaluation of the existing literature in the field of XML query optimization using semantics. We categorized the existing works into legacy databases and XML databases.

The *legacy databases* consist of the works of relational data, object-oriented, and deductive databases which utilize semantics in the schemas to optimize queries.

The *XML databases* consist of two sub-categories: XML queries without predicates and XML queries with predicates.

For the sub-category of *XML query without predicates*, we review existing techniques that utilize semantics from XML schemas for query optimization purposes in XML databases. An XML query (i.e. in XPath or XQuery) can be expressed without predicates. The works in this category are further divided into XPath query containment, tree pattern minimization and semantic query optimization. In *XPath query containment*, all techniques that apply semantics to find containment are reviewed. XPath query containment determines a set of answers of one XPath query, which is contained in another XPath query. In *tree pattern minimization*, all techniques express an XML query on a tree pattern and minimize the size of a tree pattern for query optimization purposes. In *semantic query optimization*, all works use schema semantics to formulate a set of rules, which are then used to optimize XML queries (i.e. in XPath query and XQuery).

A predicate in an XML query expresses conditions to be fulfilled in addition to the given structural path. A condition is a Boolean expression. It may involve comparisons between elements and values, path expressions denoting elements to be compared, as well as further path expressions. *XML query with predicates* review all existing techniques that use semantics in XML schemes to optimize XML queries that focus on XML query predicates.

At the end of Chapter 2, outstanding problems in these existing techniques have been identified. The scope of the work in this thesis is defined in the context of these problems.

Chapter 3 describes the problem definition of using semantics in XML schemas (i.e. DTD and XSD denoted as XML Schema or XML-Schema) to transform XML queries for optimization purposes. Chapter 3 provided a brief overview of the problem definitions.

Due to the increase in popularity of XML technology, XML Schema has become a better choice due to its richness of semantics and its variety of flexible data structures. There is a need to provide a comprehensive and systematic means of assisting database developers to exploit the great advantage of semantics contained in the XML Schema in order to transform XPath queries for query optimization purposes.

The overview also points out the problems of using semantics in XML Schemas since the types of semantics used for the important task of query optimization need to be very clear. Also, when using semantics for query transformation, the semantics are also useful for preventing certain types of queries from accessing databases. This is because these queries are identified as having conflicts and definitely return empty result sets. The overview also describes the problems of different components of XPath query such as simple path expressions, XPath axes and XPath query axes that need to be addressed depending on the different types of semantics in XML Schema.

In this chapter, formal definitions of the XML model and the XML Schema structures were provided. XML query components such as the type of path expressions, XPath query axes and XPath query conditions in predicate, structures and the notion of query processing were also defined and described. The chapter also described the details of available and standard semantics in the XML Schema, which are recommended by W3C, so that their importance can be emphasized for query transformation purposes.

Based on the semantics and types of semantics as well as the notation of query processing in XML, a summary of problem definitions was given; that is, the semantic transformations for XPath queries specified with simple path expressions, XPath queries specified with XPath axes and XPath queries specified with predicates. The semantic transformation algorithms were implemented for evaluation. A highly respectable commercial off-the-shelf Relational Database with

XML-Enabled features was chosen for data management. The most important aspect of the usefulness of the semantic transformations was the evaluation that enabled this research to identify semantic transformations as optimization devices.

Chapter 4 proposed two main streams of work. The first was the proposal of a methodology to derive the semantics in XML Schemas that prepared for semantic transformations.

The second was the proposal of semantic path transformations.

The derivation of semantics consisted of the derivation of two types of semantics:

- structural or path semantics (or constraints) produced list Q
- constraints of elements produced list C

The semantic path transformations included the following:

- semantic path expansion transformation transformed XPath queries that were specified with ‘//’ or ‘*’ to a full path;
- semantic path contraction transformation transformed XPath queries that were specified with ‘*’ to a contracted path; and
- semantic path complement transformation transformed XPath queries that were specified with ‘..’ to a contracted path or a full path.

Each semantic transformation given above was accompanied by an algorithm.

Chapter 5 proposed semantic transformation typologies for XPath queries specified with XPath axes. The semantic transformation rules were formulated by using semantics defined in XML Schemes that transformed XPath queries specified with axes including the following:

- semantic transformation for XPath queries specified with **following-sibling** or **preceding-sibling** axes;

- semantic transformation for XPath queries specified with **following** or/and **preceding** axes;
- semantic transformation for XPath queries specified with **ancestor** or/and **ancestor-or-self** axes;
- semantic transformation for XPath queries specified with **parent** axis.
- semantic transformation for XPath queries specified with **descendant** or/and **descendant-or-self** axes; and
- semantic transformation for XPath queries specified with **child** or/and **self** axes.

Each semantic transformation above is also accompanied by an algorithm.

Chapter 6 proposes semantic transformation typologies for XPath queries that are specified with predicates. The predicates are first determined by the proposed condition status determination function whereby each condition in the predicate is awarded a status with a full-qualifier or a partial-qualifier. The connectives between the query conditions ultimately determine the status of the predicate to enable either one of the following semantic transformation to be applied:

- Predicate Elimination Semantic transformation for XPath queries. This semantic transformation eliminated a predicate if it satisfied the rule condition.
- Predicate Reduction Semantic transformation for XPath queries. This semantic transformation reduced the size of a predicate by eliminating some conditions in the predicate if they satisfied the rule condition.

Each semantic transformation above was also accompanied by an algorithm.

Chapter 7 described the designs of the experiments used to evaluate the proposed semantic transformation algorithms in Chapters 4, 5 and 6. The experiments contrasted the query performance of XPath queries and their semantically equivalent counterparts.

While the first experiment contrasted the query performance of a set of designed XPath queries and their semantic counterparts based on the a real data set DBLP, the second experiment contrasted the query performance of a set of micro-benchmarks (also known as Michigan benchmarks) XPath queries and their counterparts based on the available benchmark data sets.

The elementary design in this chapter began by providing the background of experimental design that included the main objectives and evaluation strategies. The evaluation strategies focused on the implementation framework, the database platform, data sets, metrics, and the operational environment and result analyses.

The remainder of the chapter is divided into two main sections:

- Common set-up for experiments: Even though the evaluation strategy involves two experiments, each of which concerned the evaluation of specific semantic transformation formulated rules applied on a given set of XPath queries, the section detailed several parts of the strategy and decision making process that are common to both experiments. These were *Implementation Framework & Platform, Supporting of Minimal Requirement, Choice of Experiment Data and Schema, Setup of Operational Hardware, Software and System Modules*
- Individual set-up for experiments: This section described individual experiment set-ups including *semantics enhancement based on individual XML Schema (DBLP or Michigan XMLSchema), data scaling, data cleansing, query-set, metrics and procedures*.

Chapter 8 evaluates semantic transformations based on the query performance of XPath queries and their semantic counterparts using DBLP data sets. The main goal was to demonstrate the significant performance of XPath queries before and after undergoing semantic transformation. The results enable us to thoroughly evaluate the performance of XPath queries to which particular semantic transformations have been applied, but more importantly, to identify semantic transformations as optimization devices.

The evaluation is divided into three main categories:

- *Semantic path transformations*: XPath queries specified with simple path expressions were transformed by applying semantic path expansion and/or semantic path contraction or/and semantic path complement.
- *Semantic transformations for XPath axes*: XPath queries were specified with XPath axes such as child, descendant, descendant-or-self, parent, ancestor, ancestor-or-self, following, preceding, following-or-preceding sibling were transformed by the corresponding semantic rule to individual axis.
- *Semantic transformations for predicates*: XPath queries that were specified with XPath predicates were transformed by applying semantic predicate elimination semantic transformation or predicate reduction semantic transformation.

The original XPath queries and their semantic XPath queries were then run to access the database. The performance results (for semantic XPath query, performance result includes the transformation time) of the two XPath queries were compared, analyzed and evaluated. The corresponding semantic transformation was then identified as an optimization device based on the analyzed result.

The chapter also included the evaluation of those XPath queries detected with conflicts that were unsatisfied and therefore as a result, the transformation, namely *Semantic Conflict Detect* produced no semantic XPath query and the returned result produced by the original XPath query was NULL.

Chapter 9 evaluated semantic transformations based on query performance XPath queries and their semantic counterparts using the Michigan benchmark queries and data sets. This chapter complemented Chapter 8 as most of the real application data including DBLP had a very shallow data hierarchy but more branching elements that made the data structures less expressive in depth. The focus in this chapter was on the evaluation of expressiveness of data hierarchies which enabled more complex query predicates to be specified.

The query performance of XPath queries, some of which were specified with XPath axes, simple path expressions and predicates were presented. Since XPath queries in

this chapter were transformed based mainly on the presence of the predicates, the XPath query predicates were grouped under three different categories:

- *Single Query Condition with Value-based Comparison*
- *Multiple Query Conditions with Value-based Comparison*
- *Twig Pattern Query Condition with Value-based Comparison*

Under each of these categories, the XPath queries were transformed and then run to access the database. The query performance results for each pair of queries (original and semantic XPath queries) were evaluated and analyzed.

The experimental results have illustrated the majority of semantic transformations achieved a significant improvement on performance of XML query processing. Semantic transformations have been done in a more systematic manner. As the semantic transformations were done in a more systematic manner, this enabled the research here to identify semantic transformations as optimization devices.

10.2 Limitations of the Work

The outcome of every research faces some challenges and limitations. Some can be overcome, but others may require further investigation and extension left for future work. This research faces a few challenges, the major one being the *limitation of resources*, such as hardware which meant not being able to accommodate a bigger set of data for experimentation. Due to the excess research contributions, handling recursive XML Schemas is not yet addressed in thesis.

The hardware environment restricts the possibilities of testing the implementation on different platforms which require better hardware architecture. Because of the limitation of the hardware, we were not able to consider the range of data sets as significantly as planned. Chapter 7 has presented the incremental data sets used for evaluating XPath queries specified with XPath axes that have been reduced partly due to available hardware memory and database platform limitations.

When an XML Schema contained cycles, it causes recursion in XML data. The traditional approaches of matching paths or enumerating paths no longer work as it will cause an indefinite path matching issue.

10.3 Future Work

The following complementary work could address the limitations mentioned above, and at the same time broaden the scope of the current research.

- The affordability of upgrading hardware facilities with an improved availability configuration should allow the same experimentation using much larger sets of data possibly run on different platforms. This will provide more comprehensive analytical outcomes of queries, whose performance can be determined by the platform or/and engine-dependent hardware.
- Sometimes, further query conditions introduced to the predicate may enable the query to perform more efficiently. This transformation technique is known as *Semantic introduction* [Chakravarthy et al., 1986a]. The next phase of this work should explore the study of reasoning theory to extend the semantic transformation for XPath queries specified with a predicate. That is, the predicate is introduced with further query conditions.
- Among the family members of XML queries, XPath query is the most important one as it is used by most of the other family members. Semantic transformations are now recommended for application on both XQuery and XSLT to explore the optimization opportunities. In XQuery, the structure is FLWOR (For-Let-While-Or by-Return) [W3C, 2010], the semantic transformation approach should first explore the needs to determine which component to start with, before the semantic transformations apply.
- Semantically, this research focused on standard semantics that were defined in XML Schemas and recommended by W3C, 2010. Semantic transformations should ideally support semantics from alternative aspects. This research will greatly focus on ontologies in the next phase, as ontologies provide a different set of semantics that can enrich XML data structures [Sun et al., 2006].

- Capturing recursive paths in recursive XML Schema and data in order to handle the transformation of recursive XML queries would create a fresh opportunity in semantic query transformation. Again this must be conducted in a systematic manner to identify optimization opportunities.
- Exploring the existing index optimization techniques to study their usefulness. This also gives us opportunity to bridge the gap that this research cannot support in optimization. This also allows us to explore the integration opportunities of techniques to provide one complete solution which is highly efficient in terms of resources and optimization.

10.4 Closing Statement

The objective in this research was ultimately to offer a comprehensive semantic query optimization framework for XML databases. This was necessary since vendors are proposing versatile but opaque solutions that do not allow intrusive or even fine-grain optimization techniques. The vision here was one of loosely-coupled semantic optimizations that respected the vendors' autonomy, while providing significant benefits to the user.

References

- Abiteboul, S., Quass, D., McHugh, J., W., J., and Wiener, J. 1997, 'The Lorel Query Language for Semistructured Data', *International Journal on Digital Libraries*, vol. 1, no.1, pp. 68-88.
- Al-Khalifa, S., Jagadish, H. V., Koudas, N., Patel, J.M., Srivastava, D., Wu, Y. 2002, 'A Primitive for Efficient XML Query Pattern Matching', *Proceedings of 18th International Conference on Data Engineering*, pp. 141-152.
- Amer-Yahia, S., Cho, S., Lakshmanan, V. and Srivastava, D. 2001, 'Minimization of Tree Pattern Queries', *Proceedings of the ACM Sigmod Conference on Management of Data*, pp. 497-508.
- Amer-Yahia, S., Cho, S., Lakshmanan, V. and Srivastava, D. 2002, 'Tree Pattern Query Minimization', *Very Large Data Bases Journal*, vol. 11, no. 4, pp. 315-331.
- Bashir, E. and Boulos, J. 2005, 'Trading Precision for Throughput in XPath Processing', *Proceedings of the 2nd Workshop on XQuery Implementation*, pp. 552-559.
- Baqasah, A. and Pardede., E. 2010, 'Managing Schema Evolution in Hybrid XML-Relational Database Systems', *Proceedings of the 2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*, pp. 455-460.
- Bao, Z., Ling, T., Lu, J., Chen, B. 2008, 'SemanticTwig: A Semantic Approach to Optimize XML Query Processing', *Proceeding of the 13th Conference of Database Systems for Advanced Applications*, pp. 282-298.

- Brantner, M., Helmer, S., Kanne, C., Moerkotte, G. 2005, 'Full-Fledged Algebraic XPath Processing in Natix', *Proceedings of 21st Conference on Data Engineering*, pp. 705-716.
- Beyer, K. Cochrane, R. Hvizdos, M. Josifovski, V. Kleewein, J. Lapis, G. Lohman, G. Lyle, R. Nicola, M. Ozcan, F. Pirahesh, H. Seemann, N. Singh, A. Truong, T. Van der Linden, R. C. Vickery, B. Zhang, C. Zhang, G. 2006, 'DB2 Goes Hybrid: Integrating Native XML and XQuery with Relational Data and SQL', *IBM System Journal*, vol. 4, no. 2, pp. 271 – 298.
- Benedikt, M., Fan, W. and Geerts, F. 2005, 'XPath satisfiability in the presence of DTDs', *Proceedings of the 24th ACM Symposium on Principles of Database Systems*, pp. 25-36.
- Björklund, H., Martens, W., and Schwentick, T. 2008, 'Optimizing Conjunctive Queries over Trees Using Schema Information', *Proceedings of the 33rd Symposium on Mathematical Foundations of Computer Science*, pp. 132-143.
- Bohm, k., Aberer, K., Ozsu, M. and Gayer, K. 1998, 'Query Optimization for Structured Documents Based on Knowledge on the Document Type Definition', *Advances in Digital Libraries*, pp. 196-205.
- Bourret, R. 2005, *XML and Databases*,
<http://www.rpbourret.com/xml/XMLAndDatabases.htm>
- Bruno, N., Koudas, N., Srivastava, D. 2002, 'Holistic Twig Joins: Optimal XML Pattern Matching', *Proceeding of 2002 ACM SIGMOD Conference on Management of Data*, pp. 310–321.
- Ceri, S., Comai, S., Damiani, E., Fraernali, P., Paraboschi, S., and Tanca, L. 1999, 'XML-GL: A Graphical Language for Querying and Restructuring XML Documents', *Proceedings of the 8th World Wide Web Conference*, pp. 1171–1187.

-
- Chakravarthy, U. S., Grant, J. and Minker, J. 1986a, 'Semantic Query Optimization', *Proceedings of the 1st International Workshop on Expert Database Systems*, pp. 659-675.
- Chakravarthy, U. S., Grant, J. and Minker, J. 1986b, 'Semantic Query Optimization: Additional Constraints and Control Strategies', *Proceedings of the 1st International Conference in Expert Database Systems*, pp. 345-379.
- Chakravarthy, U. S., Grant, J. and Minker, J. 1988, 'Foundations of Semantic Query Optimization for Deductive Databases', in *Foundations of Deductive Databases and Logic Programming* Minker J. (ed.), Morgan Kaufmann Publisher Inc., pp. 243-273.
- Chakravarthy, U. S., Grant, J. and Minker, J. 1990, 'Logic-Based Approach to Semantic Query Optimization', *ACM Transactions on Database Systems*, vol. 15, no. 2, pp. 162-207.
- Chamberlin, D., Robie, J., and Florescu, D. 2000, 'Quilt: An XML Query Language for Heterogeneous Data Sources', *Third Workshop WebDB 2000 on the World Wide Web and Databases*, pp. 1-25.
- Che, D., Aberer, K., and Özsu, T. 2006, 'Query Optimization in XML Structured-Document Databases', *The Very Large Data Bases Journal*, vol. 15, no.3, pp. 263-289.
- Che, D.; Ling, T.; Hou, W. 2011, 'Holistic Boolean-Twig Pattern Matching for Efficient XML Query Processing', *Transactions on Knowledge and Data Engineering*, vol. PP, no. 99, pp. 1-15.
- Chen, D. and Chan, C. 2008, 'Minimization of Tree Pattern Queries with Constraints', *Proceedings of 2008 ACM SIGMOD Conference on Management of Data*, pp. 609-622.
- Clark, J. and Murata, M. 2001, *Relax NG Specification*, <<http://www.relaxng.org/spec-20011203.html>>
- Conklin, J. 1987, 'Hypertext: An Introduction and Survey', *Computer*, vol. 20, no. 9, pp. 17-41.
-

- Diao, Y., Altinel, M., Franklin, J. M., Zhang, H. and Fischer, P. 2003, 'Path Sharing and Predicate Evaluation for High-performance XML Filtering', *ACM Transaction Database Systems*, vol. 28, no.4, pp. 467-516.
- Deutsch, A., Fernandez, M., and Florescu, D. 1999, 'A Query Language for XML', *Proceedings of the 8th World Wide Web Conference*, pp. 1155–1169.
- Deutsch, A., Popa, L., and Tannen, V. 2006, 'Query Reformulation with Constraints', *SIGMOD Record*, vol. 35, no.1, pp. 65-73.
- Fan, W. 2005, 'XML Constraints: Specification, Analysis, and Applications', *Proceedings of the 16th International Workshop on Database and Expert Systems Applications*, pp. 805-809.
- Ferrarotti, F., Hartmann, S., and Link, S. 2011, 'A Precious Class of Cardinality Constraints for Flexible XML Data Processing', *Proceedings of the 30th International Conference on Conceptual Modeling*, pp. 175-188.
- Fernandez, M. F. and Suciu, D. 1998, 'Optimizing Regular Path Expressions Using Graph Schemas', *Proceedings of the 14th Conference on Data Engineering*, pp. 14-23.
- Figueira, D. 2009, 'Satisfiability of Downward XPath with Data Equality Tests', *Proceedings of the 28th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp. 197-206.
- Furfaro, S., Masciari, E. 2003, 'On the Minimization of XPath Queries', *Proceedings of 29th Very Large Database Conference*, pp. 153-164.
- Grant, J., Gryz, J., Minker, J., and Raschid, L. 1997, 'Semantic Query Optimization for Object Databases', *Proceedings of the 13th Conference on Data Engineering*, pp. 444-453.
- Groppe S. and Bottcher, S. 2005, 'Schema-based Query Optimization for XQuery Queries', *Proceeding of the 9th East-European Conference on Advances in Databases and Information Systems*, pp. 80-94.
- Groppe J. and Groppe S. 2006, 'Satisfiability-Test, Rewriting and Refinement of Users' XPath Queries According to XML Schema Definitions', *Advances in Databases and Information Systems*, vol. 4152, pp. 22-38.

- Hammer, M. and Idondik, S. B. 1980, 'Knowledge-based Processing', *Proceedings of the 6th Conference on Very Large Data Bases*, pp. 137-146.
- Hanson, P. and Mani, M. 2010, 'Semantic Optimization of XQuery by Rewriting', *Advances in Databases and Information System Workshop*, pp. 87-95.
- Haseman, W., Lin, T., Nazareth, D. 1999, 'An Intelligent Approach to Semantic Query Processing', *Proceedings of the Fifth Americas Conference on Information Systems*, pp.52-54
- IBM Corporation 2009, *DB2 pureXML*, < <http://www-01.ibm.com/software/data/db2/xml/> >
- ISO 8879-1986 Information Processing - Text and Office Systems Standard Generalized Markup Language (SGML) 1996, Geneva, Switzerland: International Standards Organization.
- Ishihara, Y., Shimizu, S. and Fujiwara, T. 2010, 'Extending the Tractability Results on XPath Satisfiability with Sibling Axes'. *Proceedings of the 7th International XML Database Conference on Database and XML Technologies*, pp. 33-48.
- Jelliffe, R. 2005, *Schematron*, <http://xml.ascc.net/schematron/>.
- Jensen, S., Plale, B., Lee, P. S. and Sun, Y. 2006, 'A Hybrid XML-Relational Grid Metadata Catalog', *Proceedings of the 2006 International Conference Workshops on Parallel Processing*, pp. 15-24.
- Jin, R., Ruan, N., Xiang, Y., and Wang, H. 2011, 'Path-tree: An Efficient Reachability Indexing Scheme for Large Directed Graphs', *Journal of ACM Transaction Database System*, vol. 36, no.1, pp. 1- 44.
- Kha, D., Yoshikawa, M. 2006, 'An Efficient Schema-Based Technique for Querying XML Data', *IEICE Transactions*, vol. 89, no. 4, pp. 1480-1489.
- King, J. 1981a, 'Quist: A System for Semantic Query Optimization in Relational Databases', *Proceedings of the 7th Conference on Very Large Data Bases*, pp. 510-517.

- King, J. J. 1981b, 'Query Optimization Through Semantic Reasoning'. *Ph.D. Dissertation*, Stanford University.
- Klarlund, N., Moller, A., and Schwartzbach, M. I. 2002, 'The DSD schema language. Automat', *Software. Engineering*, vol., 9, no.3, pp. 285–319.
- Kwong A. and Gertz, M. 2002, *Schema-based Optimization of XPath Expressions*. Technical Report, University of California. Department of Computer Science.
- Ley, M. 2011, *XML records - DBLP Computer Science Bibliography - Universität Trier*, <<http://dblp.uni-trier.de/xml/>>
- Li, J. B. And Miller, J. 2005, 'Testing the Semantics of W3C XML Schema', *Proceedings of the 29th Annual International Computer Software and Applications Conference*, pp. 443-448.
- Li, Y., Bressan, S., Dobbie, G., Lacroix, Z., Lee, M., Nambiar, U and Wadhwa, B. 2001, 'XOO7: Applying OO7 Benchmark to XML Query Processing Tool', *Proceedings of the 10th International Conference on Information and Knowledge Management*, pp. 167-174.
- Li, M., Mani, M. and Rundensteiner, E. A. 2008, 'Semantic Query Optimization for Processing XML Streams with Minimized Memory Footprint', *Proceedings of the 2008 EDBT Workshop on Database Technologies for Handling XML Information on the Web*, pp. 27-36.
- Link, S., Trinh, T. 2007, 'Know Your Limits: Enhanced XML Modeling with Cardinality Constraints', *Tutorials and Posters Contributions of the 26th Conference on Conceptual Modeling*, pp. 19-30.
- Liu, C, Millist, Vincent, V. M., and Liu, J. 2006, 'Constraint Preserving Transformation from Relational Schema to XML Schema', *World Wide Web Journal*, vol. 9, no. 1, pp. 93-110.
- Liu, Z. and Murthy, R. 2009, 'A Decade of XML Data Management: An Industrial Experience Report from Oracle', *Proceedings of the 2009 IEEE Conference on Data Engineering*, pp. 1351-1362.
- Malloy, M and Mlynkova, I. 2009, 'Closing the Gap Between XML and Relational Database Technologies: State-of-the-Practice, State-of-the-Art and Future

- Directions'. In Pardede, E. (ed.), *Open and Novel Issues in XML Database Applications: Future Directions and Advanced Technologies*, pp. 1-27.
- Meier, M., Schmidt, M., Wei, F. and Lausen, G. 2010, 'Semantic Query Optimization in the Presence of Types', *Proceedings of the 29rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems of Data*, pp. 111-122.
- Miklau, G. and Suciu, D. 2002, 'Containment and Equivalence for an XPath Fragment', *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 65-76.
- Moro, M. M., Lim, L., Chang, L. 2007, 'Schema advisor for hybrid relational-XML DBMS', *Proceedings of the 2007 ACM SIGMOD Conference on Management of Data*, pp. 959-970.
- Murata, M., Lee, D., Mani, M. and Kawaguchi, K. 2005. 'Taxonomy of XML Schema Languages Using Formal Language Theory'. *ACM Transaction. Internet Technology*. vol.5, no.4, pp. 660-704.
- Nicola, M., Kogan, I and Schiefer, B. 2007, 'An XML Transaction Processing Benchmark', *Proceedings of 2007 ACM SIGMOD International Conference on Management of Data*, pp. 937-948.
- Olteanu, D., Meuss, H., Furche, T., and Bry, F. 2002, 'XPath: Looking Forward', *Proceedings of the EDBT Workshop on XML Data Management*, 109-127.
- Oracle Corporation 2010, *Oracle Database 11g Release 2*, <<http://www.oracle.com/technetwork/database/enterprise-edition/overview/index.html>>.
- Ozcan, F., Seemann, N., Wang, L. 2008, 'XQuery Rewrite Optimization in IBMR DB2 pureXML', *Data Engineering Bulletin*, vol. 31, no. 4, pp. 25-32.
- Paparizos, S., Wu, Y, Lakshmanan, L. V. S. and Jagadish. V. H. 2004, 'Tree logical Classes for Efficient Evaluation of XQuery', *Proceeding of 2004 Conference on Management of Data*, pp.71-82.

- Paparizos, S., Patel, J. and Jagadish, V. H. 2007, 'SIGOPT: Using Schema to Optimize XML Query Processing', *Conference on Data Engineering*, pp. 1456-1460.
- Ramanan, P. 2002, 'Efficient Algorithms for Minimizing Tree Pattern Queries', *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 299-309.
- Runapongsa K., Patel J.M., Jagadish H.V., Chen Y., and Al-Khalifa S. 2003, 'Method for Efficient Storage and Indexing in XML Database', Ph.D. Thesis, University of Michigan.
- Runapongsa K., Patel J.M., Jagadish H.V., Chen Y., and Al-Khalifa S. 2006, 'The Michigan Benchmark: Towards XML Query Performance Diagnostics', *Information System*, vol.31, No. 2, pp. 73–97.
- Salminen, A. and Tompa, F.W. 1994, 'PAT expressions: an Algebra for text Search', *Acta Linguistica Hungarica*, vol. 41, no. 1, pp. 277-306.
- Salminen, A. and Tompa, Wm. 2001, 'Requirements for XML Document Database Systems', *Proceedings of the 2001 ACM Symposium on Document Engineering*, pp. 85-94.
- Schmidt, S., Waas, F., Kersten, M., Carey, J. M., Manolescu, I. and Busse, R. 2002, 'XMark: a Benchmark for XML Data Management', *Proceedings of the 28th International Conference on Very Large Data Bases*, pp. 974-985.
- Shenoy, S. T. and Ozsoyoglu, Z. M. 1987, 'Design and Implementation of a Semantic Query Optimizer', *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 3, pp. 344 –361.
- Software AG 2009, *Tamino XML Database*,
<<http://www.softwareag.com/corporate/products/wm/tamino/default.asp>>.
- Sperberg-McQueen, C. and Thompson, H. 2005, *XML Schema*,
<<http://www.w3.org/XML/Schema>>.
- Stromback, L., Asberg, M. and Hall, D. 2009, 'HShreX - A Tool for Design and Evaluation of Hybrid XML Storage', *Database and Expert Systems Application 20th International Workshop*, pp. 417 – 421.

- Su, H., Rundensteiner, E and Mani, M. 2005, 'Semantic Query Optimization for XQuery over XML Streams', *Proceedings of the 31st Conference on Very Large Data Bases*, pp. 277-282.
- Sun, W. and Liu, D. 2006, 'Using Ontologies for Semantic Query Optimization of XML Databases', *The 5th Workshop on Knowledge Discovery from XML Documents*, pp. 64 -73.
- Thomas, H., Cormen, Charles E., Leiserson Ronald L. 2001, *Introduction to Algorithms*, 2nd edn, MIT Press and McGraw-Hill, pp. 540–549.
- W3C Clark, J. 1998, *Extensible Markup Language (XML) 1.0*, W3C Recommendation, November 1998, <<http://www.w3.org/TR/1998/REC-xml-19980210>>.
- W3C Clark, J., DeRose, S. 1999, *XML Path Language (XPath) 1.0*, W3C Recommendation, November 1999, <<http://www.w3.org/TR/1999/REC-xpath-19991116>>.
- W3C Thompson, H., Beech D., Maloney, M., Mendelsohn, N. 2004a, *XML Schema Part 1: Structures Second Edition*, W3C Recommendation 28 October 2004 <<http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/>>.
- W3C Brion, P., Permanente, K., Malhotra, A. 2004b, *XML Schema Part 2: Datatypes Second Edition*, W3C Recommendation 28 October 2004, <<http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/>>.
- W3C Berglund, A., Boag, S., Chamberlin, D., Fernández, M., Kay, M., Robie, J., Siméon, J. 2007a, *XML Path Language (XPath) 2.0*, W3C Recommendation, December 2007, <<http://www.w3.org/TR/2010/REC-xpath20-20070123/>>.
- W3C Boag, S., Chamberlin, D., Fernández, M., Kay, Florescu, D., Robie, J., Siméon, J. 2007b, *XQuery 1.0: An XML Query Language*, W3C Recommendation, 2007, <<http://www.w3.org/TR/2007/REC-xquery-20070123/>>.
- W3C Berglund, A., Boag, S., Chamberlin, D., Fernández, M., Kay, M., Robie, J. and Siméon, J. 2010, *XML Path Language (XPath) 2.0, Second Edition*, W3C

- Recommendation, December 2010, <<http://www.w3.org/TR/2010/REC-xpath20-20101214/>>.
- Wang, G., Liu, M. and Yu, J. 2003, 'Effective Schema-Based XML Query Optimization Techniques', *Proceedings of the 7th Database Engineering and Application Symposium*, pp. 1-6.
- Wang, S., Su, H., Li, M., Wei, M., Yang, S., Ditto, D., Rundensteiner, A. E., and Mani, M. 2006, 'R-SOX: Runtime Semantic Query Optimization Over XML Streams', *Proceedings of the 32nd Conference on Very Large Data Bases*, pp. 1207-1210.
- Wang, J., Yu, J., Liu, C. 2008, 'Transforming Tree Pattern with DTD for Query Containment Test', *Proceedings of the 19th of Database and Expert Systems Application*, pp. 727-734.
- Wang, J. Yu, J. 2010, 'Chasing Tree Patterns under Recursive DTDs', *Proceeding of the 15th of Database Systems for Advanced Applications*, pp. 250-261.
- Wood, P. 2003, 'Containment for XPath Fragments under DTD Constraints', *Proceedings of the 9th Conference on Database Theory*, pp. 300-314
- Wu, X., Ling, T.W., Lee, M.-L., Dobbie, G. 2001, 'Designing Semistructured Databases using ORA-SS Model', *Proceedings of the 2nd Conference on Web Information Systems Engineering*, 171–180.
- Wu, Y., Patel, J.M.; Jagadish, H.V. 2003, 'Structural Join Order Selection for XML query optimization', *Proceedings of 19th International Conference on Data Engineering*, pp. 443-454.
- Wu, X., Souldatos, S., Theodoratos, D., Dalamagas, T., Sellis, T. 2008, 'Efficient Evaluation of Generalized Path Pattern Queries on XML Data', *Proceedings of the 17th Conference on World Wide Web*, pp. 835-844.
- Wu, H., Ling, T.W., Dobbie, G., Bao, Z., Xu, L. 2010, 'Reducing Graph Matching to Tree Matching for XML Queries with ID References', *Proceedings of 21st International Conference on Database and Expert Systems Applications*, pp. 391–406.

- Wu, H., Ling, T.W., Chen, B., Xu, L. 2011, 'TwigTable: Using Semantics in XML Twig -Pattern Query Processing', *Journal on Data Semantics*, vol. 6720, no. xv, pp. 102-129.
- Yao, Benjamin B., Ozsu, T., Keenleyside, J 2002, 'XBench - A Family of Benchmarks for XML DBMSs', *Proceedings of the 2002 Workshop EEXTT and Workshop DTWeb on Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web*, pp. 162-164.
- Yao, T. J. and Zhang, M., A. 2004, 'Fast Tree Pattern Matching Algorithm for XML Query', *Proceedings of the 2004 Conference on Web Intelligence*, pp. 235-241.
- Yuen, L., Poon, C. K. 2005, 'Relational Index Support for XPath Axes', *Proceedings of Third International XML Database Symposium*, pp. 84-98
- Zhang, N., Agarwal, N., Chandrasekar, S., Dacula, S., Medi, V., Petride, S, and Sthanikam, B. 2009, 'Binary XML Storage and Query Processing in Oracle 11g', *Very Large Data Bases Endow*, pp. 1354-1365.
- Zhou, R., Liu, C., Wang, J., Li, J. 2009, 'Containment between Unions of XPath Queries', *Proceedings of the 14th Conference of Database Systems for Advanced Applications*, pp. 405-420.

Appendix 1: DBLP XML Schema

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!-- edited with XMLSpy v2010 (http://www.altova.com) by Department Computer Sciene &
Computer Engineering (Department Computer Sciene & Computer Engineering) -->
<!--W3C Schema generated by XMLSpy v2010 (http://www.altova.com)-->
<!--Please add namespace attributes, a targetNamespace attribute and import elements according to
your requirements-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:import namespace="http://www.w3.org/XML/1998/namespace"/>
  <xs:element name="dblp">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element ref="article" maxOccurs="unbounded"/>
        <xs:element ref="inproceedings" maxOccurs="unbounded"/>
        <xs:element ref="proceedings" maxOccurs="unbounded"/>
        <xs:element ref="book" maxOccurs="unbounded"/>
        <xs:element ref="incollection" maxOccurs="unbounded"/>
        <xs:element ref="phdthesis" maxOccurs="unbounded"/>
        <xs:element ref="www" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
    <xs:key name="phdthesisKey">
      <xs:selector xpath="dblp/phdthesis"/>
      <xs:field xpath="@key"/>
    </xs:key>
    <xs:keyref name="supervisor" refer="phdthesisKey">
      <xs:selector xpath="dblp/phdthesis"/>
      <xs:field xpath="supervisor"/>
    </xs:keyref>
  </xs:element>
  <xs:element name="article">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="author" maxOccurs="unbounded"/>
        <xs:element ref="title"/>
        <xs:element ref="pages" minOccurs="0"/>
        <xs:element ref="year"/>
        <xs:element ref="volume" minOccurs="0"/>
        <xs:element ref="journal" minOccurs="0"/>
        <xs:element ref="number" minOccurs="0"/>
        <xs:element ref="ee" minOccurs="0"/>
        <xs:element ref="url"/>
      </xs:sequence>
      <xs:attribute name="key" type="xs:anySimpleType" use="required"/>
      <xs:attribute name="reviewid" type="xs:anySimpleType"/>
      <xs:attribute name="rating" type="xs:anySimpleType"/>
      <xs:attribute name="mdate" type="xs:anySimpleType"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="inproceedings">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
```

```

        <xs:element ref="author" maxOccurs="unbounded"/>
        <xs:element ref="title"/>
        <xs:element ref="pages" minOccurs="0"/>
        <xs:element ref="year"/>
        <xs:element ref="crossref" minOccurs="0"/>
        <xs:element ref="booktitle" minOccurs="0"/>
        <xs:element ref="ee" minOccurs="0"/>
        <xs:element ref="url"/>
    </xs:sequence>
    <xs:attribute name="key" type="xs:anySimpleType" use="required"/>
    <xs:attribute name="mdate" type="xs:anySimpleType"/>
</xs:complexType>
</xs:element>
<xs:element name="proceedings">
    <xs:complexType>
        <xs:sequence maxOccurs="unbounded">
            <xs:element ref="author" maxOccurs="unbounded"/>
            <xs:element ref="title"/>
            <xs:element ref="booktitle" minOccurs="0"/>
            <xs:element ref="publisher" minOccurs="0"/>
            <xs:element ref="series" minOccurs="0"/>
            <xs:element ref="volume" minOccurs="0"/>
            <xs:element ref="isbn" minOccurs="0"/>
            <xs:element ref="year"/>
            <xs:element ref="url"/>
        </xs:sequence>
        <xs:attribute name="key" type="xs:anySimpleType" use="required"/>
        <xs:attribute name="mdate" type="xs:anySimpleType"/>
    </xs:complexType>
</xs:element>
<xs:element name="book">
    <xs:complexType>
        <xs:sequence maxOccurs="unbounded">
            <xs:element ref="author" maxOccurs="unbounded"/>
            <xs:element ref="title"/>
            <xs:element ref="editor" minOccurs="0"/>
            <xs:element ref="publisher" minOccurs="0"/>
            <xs:element ref="year"/>
            <xs:element ref="booktitle" minOccurs="0"/>
            <xs:element ref="isbn" minOccurs="0"/>
            <xs:element ref="url" minOccurs="0"/>
            <xs:element ref="chapter" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="key" type="xs:anySimpleType" use="required"/>
        <xs:attribute name="mdate" type="xs:anySimpleType"/>
    </xs:complexType>
</xs:element>
<xs:element name="incollection">
    <xs:complexType>
        <xs:choice maxOccurs="unbounded">
            <xs:element ref="author"/>
            <xs:element ref="title"/>
            <xs:element ref="pages" minOccurs="0"/>
            <xs:element ref="year" minOccurs="0"/>
            <xs:element ref="isbn" minOccurs="0"/>
            <xs:element ref="booktitle" minOccurs="0"/>
            <xs:element ref="url" minOccurs="0"/>
            <xs:element ref="crossref" minOccurs="0"/>
            <xs:element ref="publisher" minOccurs="0"/>
        </xs:choice>
    </xs:complexType>
</xs:element>
```

```

        <xs:element ref="cdrom" minOccurs="0"/>
        <xs:element ref="ee" minOccurs="0"/>
    </xs:choice>
    <xs:attribute name="key" type="xs:anySimpleType" use="required"/>
    <xs:attribute name="mdate" type="xs:anySimpleType"/>
</xs:complexType>
</xs:element>
<xs:element name="phdthesis">
    <xs:complexType>
        <xs:sequence maxOccurs="unbounded">
            <xs:element ref="author"/>
            <xs:element ref="title"/>
            <xs:element ref="year"/>
            <xs:element ref="school" minOccurs="0"/>
            <xs:element ref="number" minOccurs="0"/>
            <xs:element ref="series" minOccurs="0"/>
            <xs:element ref="url" minOccurs="0"/>
            <xs:element ref="ee" minOccurs="0"/>
            <xs:element name="supervisor" type="xs:IDREFS"/>
        </xs:sequence>
        <xs:attribute name="key" type="xs:ID" use="required"/>
        <xs:attribute name="mdate" type="xs:anySimpleType"/>
    </xs:complexType>
</xs:element>
<xs:element name="www">
    <xs:complexType>
        <xs:sequence maxOccurs="unbounded">
            <xs:element ref="author" maxOccurs="unbounded"/>
            <xs:element ref="title"/>
            <xs:element ref="editor" minOccurs="0"/>
            <xs:element ref="year" minOccurs="0"/>
            <xs:element ref="url" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="key" type="xs:anySimpleType" use="required"/>
        <xs:attribute name="mdate" type="xs:anySimpleType"/>
    </xs:complexType>
</xs:element>
<xs:element name="author" type="xs:string"/>
<xs:element name="editor" type="xs:string"/>
<xs:element name="address" type="xs:string"/>
<xs:element name="title">
    <xs:complexType mixed="true">
        <xs:sequence maxOccurs="unbounded">
            <xs:element ref="tn"/>
            <xs:element ref="tt" minOccurs="0"/>
            <xs:element ref="ref" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="booktitle" type="xs:string"/>
<xs:element name="pages" type="xs:string"/>
<xs:element name="year">
    <xs:simpleType>
        <xs:restriction base="xs:integer">
            <xs:minInclusive value="1950"/>
            <xs:maxInclusive value="2020"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>

```

```
<xs:element name="journal" type="xs:string"/>
<xs:element name="volume" type="xs:string"/>
<xs:element name="number" type="xs:string"/>
<xs:element name="month" type="xs:string"/>
<xs:element name="url" type="xs:string"/>
<xs:element name="ee" type="xs:string"/>
<xs:element name="cdrom" type="xs:string"/>
<xs:element name="school" type="xs:string"/>
<xs:element name="publisher">
  <xs:complexType mixed="true">
    <xs:attribute name="href" type="xs:anySimpleType"/>
  </xs:complexType>
</xs:element>
<xs:element name="note" type="xs:string"/>
<xs:element name="crossref">
  <xs:complexType mixed="true">
    <xs:attribute name="href" type="xs:anySimpleType"/>
  </xs:complexType>
</xs:element>
<xs:element name="isbn" type="xs:string"/>
<xs:element name="chapter">
  <xs:simpleType>
    <xs:restriction base="xs:int">
      <xs:minInclusive value="1"/>
      <xs:maxInclusive value="30"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="series">
  <xs:complexType mixed="true">
    <xs:attribute name="href" type="xs:string"/>
  </xs:complexType>
</xs:element>
<xs:element name="ref">
  <xs:complexType>
    <xs:attribute name="href" type="xs:anySimpleType" use="required"/>
  </xs:complexType>
</xs:element>
<xs:element name="tn" type="xs:string"/>
<xs:element name="tt" type="xs:string"/>
</xs:schema>
```

Appendix 2: Michigan

Benchmarking XML Schema

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!--W3C Schema generated by XMLSpy v2010 (http://www.altova.com)-->
<!--Please add namespace attributes, a targetNamespace attribute and import elements according to
your requirements-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:import namespace="http://www.w3.org/XML/1998/namespace"/>
  <xs:element name="eNest">
    <xs:complexType mixed="true">
      <xs:sequence>
        <xs:element name="eNest" maxOccurs="unbounded">
          <xs:key name="aU1PK">
            <xs:selector xpath="//eNest"/>
            <xs:field xpath="@aUnique1"/>
          </xs:key>
          <xs:unique name="aU2">
            <xs:selector xpath="//eNest"/>
            <xs:field xpath="@aUnique2"/>
          </xs:unique>
        </xs:element>
        <xs:element name="eOccasional" minOccurs="0">
          <xs:keyref name="aU1FK" refer="aU1PK">
            <xs:selector xpath="//eOccasional"/>
            <xs:field xpath="@aRef"/>
          </xs:keyref>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="aUnique1" type="xs:anySimpleType"
use="required"/>
      <xs:attribute name="aUnique2" type="xs:anySimpleType"
use="required"/>
      <xs:attribute name="aLevel" type="xs:anySimpleType" use="required"/>
      <xs:attribute name="aFour" type="xs:anySimpleType" use="required"/>
      <xs:attribute name="aSixteen" type="xs:anySimpleType" use="required"/>
      <xs:attribute name="aSixtyFour" type="xs:anySimpleType"
use="required"/>
      <xs:attribute name="aString" type="xs:anySimpleType" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="eOccasional">
    <xs:complexType mixed="true">
      <xs:attribute name="aRef" type="xs:anySimpleType" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```