Using semantics for XPath query transformation

Dung Xuan Thi Le*

Department of Computer Science and Computer Engineering,
La Trobe University,
Bundoora, Victoria 3083, Australia
E-mail: dx1le@students.latrobe.edu.au
*Corresponding author

Stephane Bressan

School of Computing,
Law Link National University of Singapore,
117590, Republic of Singapore
E-mail: steph@nus.edu.sg

David Taniar

Clayton School of Information Technology,
Monash University,
Clayton, Victoria 3800, Australia
E-mail: David.Taniar@infotech.monash.edu.au

Wenny Rahayu and Eric Pardede

Department of Computer Science and Computer Engineering,
La Trobe University,
Bundoora, Victoria 3083, Australia
E-mail: w.rahayu@latrobe.edu.au
E-mail: e.pardede@latrobe.edu.au

Abstract: In this paper, we propose a typology of the semantic transformations for XPath queries. We focus on two main areas. The first is structural transformation for XPath query, which can be semantically contracted, expanded or complemented using structural constraints. The second is semantic qualifier transformation where the predicates, specified by [ ], in an XPath query can be eliminated or transformed. We design a set of algorithms and implement a prototype system for evaluation. We adopt two representative off-the-shelf XML data management systems to validate the effectiveness of the semantic transformations.

Keywords: XPath; semantic transformation.

Reference to this paper should be made as follows: Le, D., Bressan, S., Taniar, D., Rahayu, W. and Pardede, E. (2010) 'Using semantics for XPath query transformation', *Int. J. Web and Grid Services*, Vol. 6, No. 1, pp.58–94.

Biographical notes: Dung Xuan Thi Le graduated with a Degree in Computer Science from Swinburne University of Technology in 1993. She received her Master Degree from Latrobe University in 2006 and currently is a PhD candidate, at the same University. Her research interests include object-oriented data modelling, data warehousing and XML query performance. She has had publications in international conference proceedings and journals.

Stéphane Bressan is an Associate Professor in the Computer Science Department at the National University of Singapore and Principal Investigator at the Centre for Maritime Studies. He received his PhD in Computer Science in 1992 from the University of Lille. In 1990, he joined the European Computer-industry Research Centre (ECRC). From 1996 to 1998, he was Research Associate at the Sloan School of Management of the Massachusetts Institute of Technology (MIT). His research interest is the integration and management of information

David Taniar holds Bachelor, Master, and PhD Degrees – all in Computer Science, with a particular specialty in databases. His current research interests include mobile/spatial databases, parallel/grid databases, and XML databases. He recently released a book: High Performance Parallel Database Processing and Grid Databases (John Wiley & Sons, 2008). His list of publications can be viewed at the DBLP server (http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/t/Taniar:David.html). He is a founding Editor-in-Chief of Mobile Information Systems, IOS Press, The Netherlands. He is currently an Associate Professor at the Faculty of Information Technology, Monash University, Australia.

Wenny Rahayu is an Associate Professor at the Department of Computer Science and Computer Engineering LaTrobe University. Her research areas cover a wide range of advanced databases topics including XML databases, spatial and temporal databases and data warehousing, and semantic web and ontology. She is currently the Head of Data Engineering and Knowledge Management Laboratory at La Trobe University.

Eric Pardede completed his Doctor of Philosophy in Computer Science and Master of Information Technology from La Trobe University. He has published his research works in various books, international journals and conference proceedings. Currently, he is a Lecturer in Software Engineering and Database at La Trobe University, Melbourne, Australia.

1 Introduction, motivation, contribution and framework

Semantic query transformation (Hammer and Jdondik, 1980; King, 1981; Shenoy and Ozsoyoglu, 1987; Deutsch et al., 2006) is the process of rewriting, with the knowledge of some integrity constraints, a query into an equivalent one. While the transformed query under semantic query optimisation aims to achieve a better performance – such as more efficient processing – the transformed query under semantic query transformation aims to achieve a better query structure and may provide some opportunities for optimisation. The concepts have been popularised for relational, object-oriented, deductive and even object-relational databases; hence, it is also useful for XML databases, especially with the current availability of rich semantics in XML schema. This has renewed interest in the study of semantic query transformation and optimisation for XML query languages such as XPath, XQuery and possibly the optimisation of programs in XML languages such as XSL.

Semantic query optimisation for XML queries has been studied by Deutsch et al. (2006), Su et al. (2004, 2005) and Sun and Liu (2006). The Chase and Back (C&B) algorithm (Deutsch et al., 2006) was suggested for reformulating XPath query by using the relational encoding and constraints in a Document Data Type (DTD) schema. The second contribution (Su et al., 2004, 2005) was concerned with XQuery for stream databases where a pruning tree was proposed for structural rewriting.

Earlier work on XML query optimisation focused extensively on pruning and minimisation (Amer-Yahia et al., 2001; Su et al., 2004, 2005; Sun and Liu, 2006; Wood, 2001, 2003) using tree patterns. A tree pattern that presents an XPath query is known as a query tree pattern (Al-Khalifa et al., 2002). One of the most popular and efficient techniques for processing a query tree pattern is known as pattern matching (Smiljanic et al., 2005). Certain structural constraints such as edges representing relationships between tree nodes are applicable.

XPath queries also allow predicates (Gupta and Suciu, 2003). However, predicates (Le and Pardede, 2009) in XPath queries are more complex than predicates in relational queries. They not only allow relationships between the tree data nodes, but also they allow values to restrict the content of elements. Very often, users specify XPath queries that unintentionally create redundancies, which could lead to a performance issue.

The **motivation** behind our work is the importance of tree pattern and rich data semantics available in the XML schemas, which have renewed rewriting opportunities. The XPath expressions can be, for instance, contracted or expanded with the knowledge of the structural constraints of XML schema. For example, the XPath query "a/b/c" can be contracted to "//c" if a schema indicates that c is always immediately preceded by a/b or vice versa; that allows the path "//c" to be expanded into "a/b/c" using the same knowledge.

With the knowledge of structural and occurrence (Biron et al., 2004) constraint, for example, the XPath query "a/b[c/d]/f" can be contracted to a path "a/b/f". That is, only when a schema indicates that c and f are always immediately preceded by a/b and the minimal occurrence of c under b must be at least 1, the same as the minimal occurrence of d under c. Such constraints allow the elimination of [c/d] to produce "a/b/f". In any XML schema (XSD), apart from structural or identity constraints (Biron et al., 2004), other constraints are considered very useful for semantic transformation categorised as the faceting constraints (Biron et al., 2004). This is a type of constraint applicable to leaf-node elements. For instance, an XPath query "/a/b[. ='valueA' or . ='valueB']"; this XPath query has a predicate [. ="valueA" or .="valueB"]. This predicate can be removed if the values {valueA, valueB"} are the exact matched values of element b specified in the associated schema (XSD) within an XML document.

Another interesting study we discover for semantic transformation is when an XPath query uses a context sensitive function (Berglund et al., 2007) such as position(), last(), first(). We find that certain constraints can be useful for transforming XPath expressions where a context-sensitive function in an XPath query is eliminated to boost the performance. For instance, an XPath query 'a/b[//d[position()>=1]]/f' indicates a nested predicate [position()>=1] with the outer predicate being [//d]. This XPath query simply lists all the 'f' information where each 'b' is the ancestor and it must have at least one 'd' as descendant. To transform this type of query, we first look at the path '//d' in the outer predicate to determine whether the path is valid and therefore it would be better to use '//', or whether '//' can be transformed to a full conditional path 'c/d'.

To transform both predicates, we must employ the structural constraint as well as the occurrence constraint between 'b' and 'c', as well as between 'd' and 'f'. If verification indicates that the constraints in the schema conform to the constraints used in XPath query for those mentioned elements, and if the minimal occurrence of 'd' under 'c' and 'c' under 'b' is 1, then a possible semantic XPath query is 'a/b/f'.

To the best of our knowledge, the techniques for semantic transformation described above have not as yet been addressed for XPath query. Although our literature survey indicates that numerous works have tackled XML query optimisation (refer to Table 1) with the presence of schema, none of the existing works overlaps our techniques.

Table 1 Summary of semantic query transformation/optimisation

Author(s)	*Approach*	*Schema constraints*
Amer-Yahia et al. (2001)	Minimisation of Tree Pattern Query (TQP)	Subtype, required child and required descendants
Ramanan (2002)	MinimiseChase	Subtype, required child and required descendants
Chen and Chan (2008)	Minimisation of Tree Pattern Query (TQP)	Subtype, required child, required descendant, required parent, required ancestor and sibling constraint
Sun and Liu (2006)	Ontology conceptualisation	Object Oriented semantics in the XML schema
Su et al. (2004, 2006) and Li et al. (2008)	Tree query technique	Structural constraints apply to the traversal of the Tree Pattern Query to determine predicates that does not contribute to the final result
Coen et al. (2004) and Wang et al. (2003)	Path complementing and path shortening	Use schema labelling to derive schema paths

Our **contribution** is as follows:

1 We derive a typology of semantic path transformations (Le et al., 2007) including *semantic path expansion*, *semantic path contraction* and *semantic path complement* to transform an XPath query that has no predicate.

2 We introduce another typology of semantic predicate transformation that consists of *semantic content-based predicate*[1] and *semantic index ordinal predicate*[2] to transform the predicates in an XPath query.

3 We then design two *algorithms* for semantics pre-processing preparation and our proposed semantic transformation typologies.

4 The evaluation of these proposed transformations will be carried out by *comparatively evaluating* the *performance* of the workload of XPath queries and their transformations. We use two systems representative of the state-of-the-art XML database management and query processing systems: a *Native XML Database System* (referred to as **XMS**) and an *XML-Enabled Relational Database System* (referred to as **XDB**). Finally, we *discuss* and *analyse* the results and outline the future work.

Below is the summary of our **framework** to achieve the contribution described above.

- First, as a criteria for structural query information, we study the path fragment [/, //, …, *].
- Second, as criteria for structural query expression, we revisit the notion of pattern matching (Al-Khalifa et al., 2002) alignment with structural constraints, which makes a significant impact on our semantic transformations.
- Third, we look into a series of constraints from the XML schema (Biron et al., 2004) to select the most useful constraints that are imperatively suitable for our semantic transformations. For transforming an explicit structural expression, the semantic path transformation, constraints/semantics ('/' parent-child, '//' ancestor-descendant) are used. To optimise the performance of the semantic path transformation typology, we integrate the notions of *pattern matching* and *canonical rewriting* with *semantic path expansion*, *semantic path contraction* and *semantic path complement*.
  For transforming predicates, we study the facet constraints (Biron et al., 2004) for data element to support our proposed semantic qualifier transformation typology including *semantic content-based predicate* that holds the leaf-node with values for filtering; and a *semantic index ordinal predicate* for predicates that hold the position(), last() or an index.
- Four, we provide two main *algorithms*. The first algorithm plays an important role in preparation such as pre-processing the schema information and efficiently storing it in well-designed data structures so that information can be easily obtained and used by the second main algorithm. This second algorithm is central to the work of this paper. Not only are all the proposed semantic transformations implemented in this algorithm, but also the algorithm is able to determine whether the XPath query returns an answer well before it is transformed or even sent to the database for query. Such a query indicates a conflict in semantics, such problem also studied by satisfiablity (Groppe and Groppe, 2006), either in structural expression or data content. The success of the semantic transformations of queries depends considerably on the derivation of such an algorithm.
- We evaluate the potential of these transformations by *comparatively evaluating* the *performance* of the workload of XPath queries and their transformations. To achieve better confidence in comparative results, we use two systems representative of the state-of-the-art XML database management and query processing: a *Native XML Database system* (we refer this to as **XMS** through out this paper) and an *XML-Enabled Relational Database system* (we refer this as **XDB**).
- Finally, we present the conclusion and the future roadmap for this work.

The semantic transformation work is considered to be a studied and traced path to semantic optimisation for query processing dealing with XML databases. Its main goal is not immediate optimisation, but rather to achieve an alternative way of writing the XPath query. Some transformations may produce very promising improvement, thereby providing opportunities for semantic query optimisation.

This paper significantly extends the existing work (Le et al., 2007), which was introduced to address only semantic path transformation using a unique path, by more than 90%. The remainder of this paper is organised as follows: Section 2 is an overview of related work on semantic query optimisation in general and of semantic query optimisation for XML query. Section 3 sheds light on the terminologies, problems and notions of query pattern; Section 4 defines the typology of semantic transformations; Section 5 describes the implementation and evaluation system including the algorithms; Section 6 discusses the results and analyses the comparative empirical performance evaluation. Lastly, we summarise our findings and outline future work in Section 7.

2 Related work

While semantic transformation is the process of rewriting all given queries despite the presence of data in the database, the status of performance of the transformed query is not determined before the query is transformed. Unlike semantic transformation, semantic optimisation is the process of selectively rewriting a query, which is undertaken only if the status of the performance can be determined before the query is transformed. The earlier contribution to this area was not able to clarify the difference between the two concepts, thereby producing misinterpretation.

Semantic notation proposed for processing queries made an initial contribution to the research area of semantic query optimisation (Hammer and Jdondik, 1980; King, 1981). Although these contributions are the conceptual frameworks for semantic query optimisation, they were extended to the design of a broader range of methodology and rules that have grown and developed quickly as a result of later works. A typology for semantic query rewriting (Charkravarthy et al., 1990) has been dispersed including *literal insertion, literal elimination, and range modification* (modification of a condition). The work has acknowledged the important role of semantic query rewriting. That is, some queries can be answered without having to access the database (typically when a contradiction is exposed in the query, which, consequently, denotes an empty answer).

Literal insertion and elimination techniques have become more common practice. Queries are formed with some simple predicates; therefore, they are easily determined for the removal of redundant checking conditions or to introduce more restrictions to avoid the use of disproportionate search space. On the other hand, range modification is an exhausting process, which may become a recursive problem. Due to this reason, we propose to study the literal insertion and literal elimination and apply them in XML queries.

Minimisation of tree pattern queries was introduced by Wood (2001) and Amer-Yahia et al. (2001). Although their work did not utilise any integrity constraints in its minimisation technique, it later drew the interest of, and was further investigated by Ramanan (2002). These works introduced integrity constraints to improve query run-time of tree pattern queries.

Two different proposed techniques including *Minimisation* and *MinimisChase* used the same integrity constraints. *Minimisation* (Amer-Yahia et al., 2001) of Tree Pattern Query (TPQ) was proposed to improve the query run-time and included integrity constraints. Subtype, child or descendants were utilised in this technique. With respect to the number *n* nodes regarded as a query size, their algorithm produced a run-time $O(n^6)$.

The later work proposed the *MinimiseChase* technique (Ramanan, 2002) which improved the run-time, achieving $O(n^4)$ in the presence of the same set of integrity constraints. Further to Ramanan's (2002) investigation, this work took the additional step of eliminating the subtype constraint from the algorithm. This left only what was considered the required child or required descendant, and their algorithm achieved a better run-time of $O(n^4)$. However, both techniques (Amer-Yahia et al., 2001; Ramanan, 2002) had shortcomings. Firstly, they considered integrity constraints in optimisation, which is only part of the whole optimisation. Secondly, they focused on structural constraints and thirdly, even with their consideration of structural constraints, neither technique was able to handle any constraints other than three: subtype, required child and required descendants.

To overcome the problem of integrity constraints limitation in the minimisation of TPQ (Chen and Chan, 2008) explored the minimisation problem of TPQ further for a class of FBST-constraints including forward (required child, required descendant), backward (required ancestor and required parent), subtype and sibling constraints. To deal with the backward and sibling constraint, their *ChaseMinimiseFBST* algorithm computes a single minimal query of an input query *Q* with respect to a set of FBST-constraints and it achieved a run time of $O(n^3|\Sigma|)$ where n is query size and Σ is a set of distinct element types in the constraints. Although this work has increased the number of structural integrity constraints to be considered for the minimisation of TPQ, and the run-time was better than that of earlier works (Amer-Yahia et al., 2001; Ramanan, 2002), the obvious shortcoming is the inability to support anything more than those integrity constraints that are already listed. Their limitation of constraints, which recognised those distinct element types, means no recursive type support.

Optimising XQuery using Semantic query optimisation was part of the proposed optimisation work for RainDrop (Su et al., 2006; Li et al., 2008) and R-SOX (Wang et al., 2006). Their goal was to use schema constraints to deal with query pattern in predicates, which they believe do not contribute to the final result. The problem with this work is that the technique is tight-coupled DTD schema. Other than structural constraints, constraints of data elements are not part of their semantic optimisation work.

Let us consider a query '//auction[reserve or bidder/zipcode contains "9000"]'.

This existing work provides no discussion about the constraint (for example a zipcode must be between 1000–6000) that can determine how such a given value in the predicate can cause a null result set. This would contribute to the performance if the query has to access a large database in order to receive the result. This is due to their focus mainly on structural constraints to expedite the traversal of the steam to avoid unnecessary computation of the final result. Providing support for constraints of data elements is not within the scope of their work.

In addition to the aforementioned limitations of this work, although it proposed to deal with semantic optimisation, the rules/techniques for rewriting queries were not clearly discussed, apart from the rules for making decisions about the parts of XQuery to be executed in order to achieve a better performance.

As a result of our survey, we summarise the outstanding issues as follows:

- The utilisation of XML schemas constraints (Su et al., 2004, 2005; Che et al., 2006; Sun and Liu, 2006) in either semantic query optimisation or transformation, as previous works focused mainly on structural constraints for tree expression. Rewriting of predicates was not significantly addressed.

- Our work is different from the existing work in that we consider the semantic transformation from two perspectives: the first is the semantic transformation for a structural (tree expression) model; the second is the content data model.
  The technique for the content data model is inherited from the semantic transformations proposed for relational databases. Some common and obvious transformations performed earlier such as *range modification*, *literal insertion*, and *elimination* (Charkravarthy et al., 1990; Shenoy and Ozsoyoglu, 1987; King, 1981; Hammer and Jdondik, 1980) have been found to be very useful for our work.
  We adopted some of these as they provide the capability to deal with data values in predicates. Although not all can be adopted, we consider and apply selective techniques to our transformations. XPath query can be semantically *transformed* when the *constraint* is *detected* and *matched*. We show that even with the process of checking existing constraints, some transformed queries perform better than the original ones.

3 Terminologies, studied problems and notions of query tree pattern

Though although information about XML documents, schema(s) and constraints is generally available and can be found in any general XML materials, the type of information required for our work needs to be streamlined. Therefore, in this section we simplify the information including XML schema, XML document and related information that is required for our work. We then describe how the XML document or database must comply with an associated XML schema. We define several important terms and notions that play key roles in our semantic transformations proposed in Section 4. One of the important roles of path expression is the query tree pattern in which pattern matching is fundamental to processing a TPQ. For this reason, the notion of pattern matching is also revisited; we present a problem and discuss examples in detail in order to illustrate its importance, since it is required by our later work.

3.1 XML schema, XML data documents and semantics/constraints

For convenience and a wider variety of semantics, we use XML schema type (XSD); this does not mean that we restrict the schema type (DTD).

Definition 1 (XML Schema)**:** An XML schema is a directed graph G that is represented by $\{\mathcal{V}, \mathcal{E}, C, \varphi\}$ where:

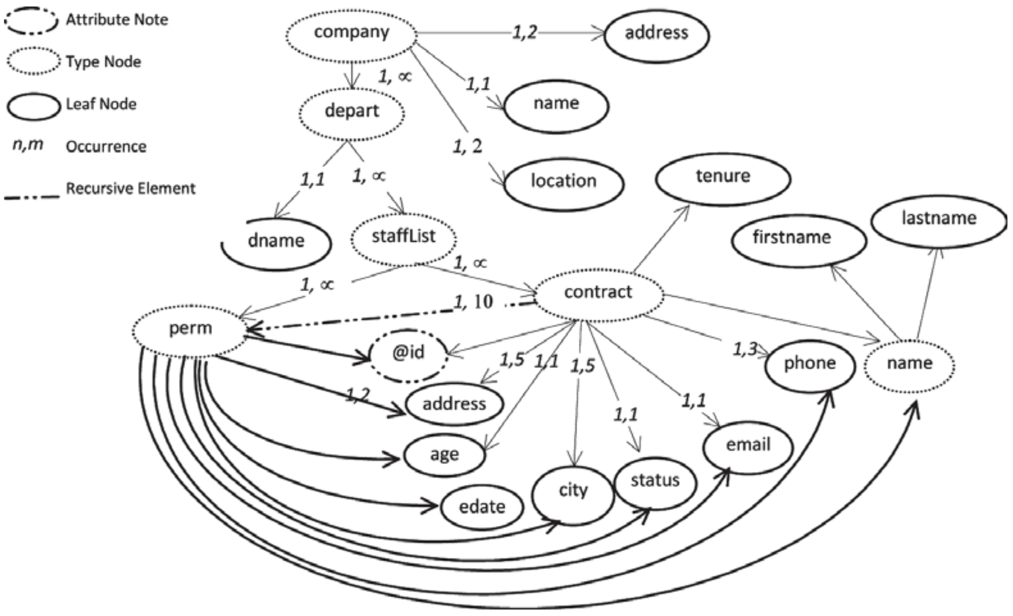
- $\mathcal{V}$ is a non-empty finite set of vertices
- $\mathcal{E} \subset \mathcal{V} \times \mathcal{V} \times I \times (I \cup \{\infty\})$, is a finite set of edges with a multiplicity (j, k) associated with each edge where j and k are integers such that $j \leq k$, $j \in I$ and $k \in I \cup \{\infty\}$
- $\varphi \in \mathcal{V}$ is the root.

The examples (but not all) below illustrate the notations based on Figure 1.

- $\mathcal{V}$ = {*company*, *depart*, *name*, *location*, *@id*}
- $\mathcal{E}$ = {(*company*, *depart*, *1*, ∞), (*depart*, *name*, *1*, *1*), (*depart*, *staffList*, *1*, ∞), (*company*, *name*, *1*, *1*), (*company*, *location*, *1*, *3*)}
- φ = *company*.

When the minimal and maximal occurrences, denoted by (*minOccur, maxOccur*), of an element are not defined in the schema; the default values are 1. For example, each contract staff member can have only one occurrence of the 'age' element.

Figure 1 Company XML schema



Definition 2 (XML Document) XML document is a tree $\mathcal{T}$ where

- *Every vertex is labelled as element or attribute or text content* (*labels and text content – can be empty*)
- *Edges* $\mathcal{E}$ *are pairs of* $\mathcal{V}$.

An XML document $\mathcal{T}$ is valid with respect to an XML schema $\mathcal{G}$ if and only if:

- the root of the XML document tree $\mathcal{T}$ is a unique node in $\mathcal{V}$ such that root φ of $\mathcal{G}$ is also root φ of $\mathcal{T}$
- there exists a total mapping $\mathcal{M}$ of nodes $\mathcal{V}$ in $\mathcal{T}$ to nodes $\mathcal{V}$ in $\mathcal{G}$ that holds the following conditions:
- a node $v \in \mathcal{V}$ in $\mathcal{T}$ is mapped to $v \in \mathcal{V}$ in $\mathcal{G}$; if v has some children then v must be a complex type; or

- a node $v \in \mathcal{V}$ in $\mathcal{T}$ is mapped to $v \in \mathcal{V}$ in $\mathcal{G}$ if v has no children and v is a simple type then it is either an attribute or a leaf-node
- for each edge, in schema directed graph $\mathcal{G}$, between two vertices (v_h, v_i) if j and $\mathcal{K}$ are the minimum and maximum occurrences of child element v_i then the number of edges for this pair of nodes is k times in XML document $\mathcal{T}$ where $0 \leq j$ and $j \leq k$
- for node $v \in \mathcal{V}$ if u found within a complex type v then u is called a sub-element of v and for any $r \in \mathcal{V}$ if r, as an attribute to itself, is found within v then r is an attribute element, there is a certain parent-child, an essential structural constraint, edge from v to u; v to r
- for any element $v \in \mathcal{V}$ if z is found to be a value v with then v is called a leaf element; z is the context content of v. Each v may be assigned with some content-based constraint[1] $c \in \mathcal{C}$.

3.2 Path and path expression

A schema path specifies the selection of vertices. The structural paths are expressed based on the XML schema and XML document. Hence, we define the class of path expression as:

Definition 3 (Path in schema $\mathcal{G}$)**:** *A* path in the XML schema $\mathcal{G}$ is a sequence of vertices ($\mathcal{V}x$, …, $\mathcal{V}z$) s.t. $\mathcal{V}z$ must occur under $\mathcal{V}x$ via $\mathcal{V}y$ where $x \leq y \leq z$.

Definition 4 (Path expression)**:** A path expression in an XML schema $\mathcal{G}$ is a path that has elements represented by '*', '..' or '.'. Elements are separated by '/' or '//'.

- // represents ancestor and descendants
- / represents parent and child
- * represents a union of vertices/elements
- . represents the vertices/elements
- .. represents the parent.

The following are (some, not all) path expressions in the XML schema:

{company//name; company//location; //depart; company/*/@id; company/*/name; company/depart; company/depart/@id; company/depart /name; company/depart/staffList, //depart/staffList /perm/../contract; */*; *//*; */*/*; etc…}.

Definition 5 (A Path in XML Document)**:** A path query over an XML document tree is a path expression if and only if

- A path and path expression are identical
- Or we replace'*' with some vertex and the path matches the new path expression
- Or we replace '.' with the current vertex then the path matches the new path expression

- Or we replace // with a sequence of nodes separated by / the path matches the new path expression
- Or we replace '..' with '[ ]' where [ ] contains a vertex or a path that matches the new path expression.

Motivation example: with reference to Figure 1 XML company schema, it can be explained as follows:

The root is the 'company' where there must be at least one department, valid location, and address. There is no limit to the number of departments but the company can be located in two locations attached to two different addresses respectively. Every department 'depart' occurs under the 'company' and must have at least one immediate department name 'dname' and a list of staff 'StaffList'. Every staff list contains information about at least one staff of permanent and contract type with no restriction on the number of staff in each list. The record of every permanent and contract staff member contains the following information: identification, name that includes first name and last name, age, address, city, email, status and phone number and employment date. Every contract staff member has, in addition, a record of duration of tenure. Staff Identification is modelled as the unique attribute 'id' in both 'perm' and 'contract' elements. The 'name', 'age', 'status', 'employment date', 'address', 'city', 'email' and 'phone' elements are child elements in both 'perm' and 'contract'. 'Age' of staff must be between 23 and 55. A staff member can be a supervisor who will have a supervision id that is restricted to an alphanumeric pattern. The company's locations are restricted to 'Bundoora' or 'Melbourne'. Every employee is assigned with a status 'Active' or 'Inactive'.

3.3 Notion of query tree patterns

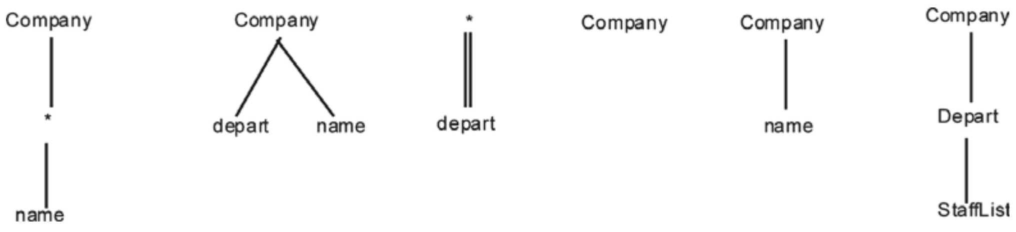
Prior to presenting our proposed semantic path transformation, we revisit the theory of query tree pattern that represents a structure of an XML query, specific to XPath query. The processing of a query tree is called pattern matching which we have also described.

Definition 6 (Query Tree Pattern)**:** A query tree pattern Q_T is a tree where every node is labelled with a vertex $u \cup *$; a wildcard * represents any tag. The semantics between two vertices u in the Q_T is represented by '/' or '//' where '/' represents a structural semantic of a parent-child and '//' represents a structural semantic of ancestor-descendant.

A query tree pattern Q_T corresponds to an XPath query $\mathcal{P}$. See the running examples of XPath queries *a*, *b* and *c* and unique paths (Le et al., 2007) *d*, *e* and *f* below for tree pattern representations in Figure 2:

XPath queries:	(a) company/*/name; (b) company[depart]/location; (c) *//depart
Unique paths:	(d) company; (e) company/name; (f) company/depart/staffList.

Figure 2 Tree paterns for XPath queries and Unique paths



Each of these query tree patterns exhibits only two types of structural relationship. While the former one ‘|’ is called a parent-child relationship which is represented by the path operator ‘/’, the latter one ‘||’ is known as an ancestor-descendant relationship.

In our running example of XPath query *b*, a query tree pattern shows that it has outgoing branches at a degree greater than 0 at the branching node ‘company’.

Definition 7 (Branching node)**:** is a node that has at least two child nodes.

The node selection by a query tree pattern ($\mathcal{Q}_{\mathcal{T}}$,) returns a non-empty valid result if, and only if, $\mathcal{Q}_{\mathcal{T}}$ is matched to a whole or portion of the XML schema graph $\mathcal{G}$. This notion is known as pattern matching.

The pattern matching (Al-Khalifa et al., 2002) notion has been studied for a query tree pattern $\mathcal{Q}_{\mathcal{T}}$ in XML database $\mathcal{T}$. However, due to the nature of our research, the XML schema is the main asset and therefore it does not access the XML database in order to decide whether the query pattern is valid; instead it depends on the schema to decide the valid ability of the query pattern. Hence, we match $\mathcal{Q}_{\mathcal{T}}$ to the paths in the schema. We revise a pattern matching definition based on our work content as below.

Definition 8 (Pattern matching)**:** A matching of a pattern query $\mathcal{Q}_{\mathcal{T}}$ is a mapping δ from $\mathcal{Q}_{\mathcal{T}1}$ (u) to $\mathcal{Q}_{\mathcal{T}1}$ (v) such that δ:

- For each u in $\mathcal{Q}_{\mathcal{T}}$, the predicate node label of u is matched by $\delta(v)$ in $\mathcal{G}$
- For each edge (u, ω) in $\mathcal{Q}_{\mathcal{T}}$, $\delta(\omega)$ is a child or descendant of $\delta(u)$ in $\mathcal{G}$
- For each edge (u, ω) in $\mathcal{Q}_{\mathcal{T}}$, *if* $\delta(u)$ is branching node then it must also is a branching node in $\mathcal{G}$.

We have revisited comprehensive material, information and background of query tree and pattern matching for XML query, schema and path structures, which we believe are useful for the readers to follow our proposed semantic query transformation. The section below presents the main work of this paper.

4 Semantic transformation typology

The concept of our unique path introduced in Le et al. (2007) is a schema labelling concept (Wang et al., 2003). For this particular work, we consider that the schema nodes are uniquely labelled among themselves. This means that a child node is recursively under multiple parents; however, a parent is restricted to not having two different children with the same label name.

Definition 9 (Unique path)**:** Unique paths Q is a set of all possible paths Q_u that are expressed over an XML schema, that contains a sequence of elements (e_x, …, e_z), s.t. e_x must be the root r of schema, e_z must occur under e_x via e_y (where $x \leq y \leq z$), and hierarchical relationships '/' parent-child expresses the relationships among the elements.

In the running example of XML company schema (Figure 1), the possible (but not all) unique paths are:

{company; company/name; company/location; company/depart; company/depart/name; company/depart/staffList; company/depart/staffList/perm; company/depart/staffList/perm/@id; etc..}

Each unique path is represented by a query tree pattern where the tree pattern strictly has no branches. A series of unique paths goes through a normalisation process, similar to the BCNF technique, to form a new query tree pattern that may allow some branches. We define such a query pattern as follows:

Definition 10 (Normalised Pattern)**:** A normalised pattern results from multiple unique paths, which are needed to produce a required answer, being replaced with '//'.

For example, we have identified two unique paths Q_{u1} and Q_{u2} which are 'company/depart/staffList/perm' and 'company/depart/staffList/contract'. A new query pattern can be formed as we can normalise these two query patterns to derive a single query tree pattern. We can remove 'company/depart' from Q_{u1} or Q_{u2} and allow the branching out on 'staffList' node that has two children 'contract' and 'perm'. This means either 'contract' or 'perm' is a conditional element in the predicate.

Not all the required answer needs multiple unique paths to produce it. A unique path is sometimes an XPath query; in which case, it has the following unique characteristics – it is based on the structure of the schema, it must start with the root vertex, and it allows only the parent-child '/' relationship. A unique path is therefore also a query tree pattern.

Any node or vertex in the XML schema can be a target[3] node. The distinct property of the unique paths is that the target node in multiple unique paths may have the same tag name; however, path structures are uniquely different in terms of parents and ancestor although they must start from the root.

4.1 Semantic path expansion transformation

The difference between the earlier *semantic path expansion* (Le et al., 2007) and this *semantic path expansion* is that while the former adopted the unique path concept, in the latter we continue using the unique path concept and ensure the accuracy of this transformation by integrating the notion of query tree pattern matching from an XPath query to one or more unique path(s). The normalisation will be applicable for semantic path contraction and semantic path complement.

Definition 11 (Semantic path expansion)**:** A semantic path expansion $\mathcal{P}'$ is an XPath query that is transformed from an XPath query $\mathcal{P}$ and matches a unique path, which produces the same result set as $\mathcal{P}$ does.

Proposition: *$\mathcal{P}'$ is a semantic expanded path if and only if $\mathcal{P}'$ is an exact match to one and only one unique path; hence, there exists only a parent-child relationship between each pair of elements in the path.*

Given a valid XML schema graph $\mathcal{G}$, we derive a set of unique paths $Q = \{Q_{u1}, Q_{u2}, Q_{u3}, \ldots, Q_{un}\}$ where $n > 1$. Each Q_u has a query tree pattern ($\mathcal{V}_Q$, $\mathcal{E}_Q$) where $\mathcal{E}_Q$ is a set of edges '/' separating set of nodes $\mathcal{V}_Q$ in Q_u.

The 'if' direction in the proposition says for an XPath query $\mathcal{P}$ that has a set of elements and edges where elements labelled with l *or '*' or '.' and edges expressed by '|' or '||'. If all elements in a Q_u are a superset of all elements in $\mathcal{P}$; '*' or '.' in $\mathcal{P}$ can be mapped to elements in Q_u and edge trees '||' (if there exists any) can be replaced by a sequence of elements labelled l separated by '/' in Q_u then Q_u patterns $\mathcal{P}$, hence Q_u is $\mathcal{P}'$.*

Rule 1: *XPath P is semantically expanded only if the following required criteria are met*:

- *Start from the target element $v_j \in \mathcal{P}$ mapped to target element $v_j \in Q_u$*
- *Map all nodes $v_i \in \mathcal{P} \rightarrow v_i \in Q_u$ where v_i is the parent or ancestor of v_j*
- *Replace all '*' and '.' in $\mathcal{P}$ with $\exists v_i \in Q_u$*
- *If '//' exists in $\mathcal{P}$, replace it with a fragment in Q_u, that has an ancestor and a descendant enclosed '//' in $\mathcal{P}$.*

Let us consider an example of XPath query $\mathcal{P}$ = company/*//perm. (1)

Referring to Figure 1, a unique path Q_u that leads to 'perm' element is 'company/depart/staffList/perm'.

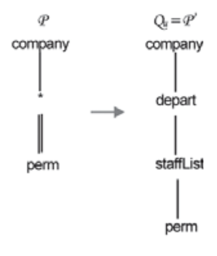
By applying *Rule 1* to $\mathcal{P}$, we explain the derivation of $\mathcal{P}'$ from $\mathcal{P}$, as depicted in Figure 3. The matching elements of $\mathcal{P}$ to elements in Q_u can be summarised such that target node 'perm' $\in \mathcal{V}_\mathcal{P} \rightarrow$ 'perm' $\in \mathcal{V}_{Qu}$ and $\mathcal{V}_{Qu} \subseteq \mathcal{V}_\mathcal{P}$; '*' $\in \mathcal{V}_\mathcal{P} \rightarrow$ 'depart' $\in \mathcal{V}_{Qu}$. The tree edge '/' represents a parent-child relationship between 'company' and '*' in $\mathcal{P}$ or 'company' and 'depart' in Q_u. As the result, we can replace '*' with 'depart', which makes up a new path 'company/depart//perm'. The tree edge '//' in $\mathcal{P}$ represents an *ancestor-descendant* where 'perm' is descendant and 'depart' is ancestor. If 'depart//'perm' can be mapped to a path fragment in Q_u where only a parent-child relationship exists between the pair of elements, then $\mathcal{P}$ can semantically be replaced by Q_u, hence, Q_u is automatically $\mathcal{P}'$.

Figure 3 $\mathcal{P}$, Q_u & $\mathcal{P}'$

$\mathcal{P}$ = company/*//perm
$\mathcal{V}_\mathcal{P}$ = {company,*,perm}
$\mathcal{E}_\mathcal{P}$ = {/,//}

Q_u = company/depart/staffList/perm
$\mathcal{V}_{Qu}$ = {company, depart, staffList, perm}
$\mathcal{E}_{Qu}$ = { / }

$\mathcal{P}'$ = company/depart/staffList/perm



In this section, we show how an XPath query is semantically transformed using the semantic path expansion rule that employs the parent-child constraint to eliminate the ancestor-descendant representative in the XPath query. The next semantic transformation rule is the *semantic path contraction*.

4.2 Semantic path contraction transformation

Path contraction is applicable only when we deal with query answers that need multiple unique paths to produce them.

Definition 12 (Semantic path contraction)**:** A semantic path contraction $\mathcal{P}'$ is transformed from an XPath query $\mathcal{P}$. $\mathcal{P}$ matches with multiple unique paths Q_u, which are required to produce the same result as $\mathcal{P}$ does.

Proposition: *Some path fragments in $\mathcal{P}'$ are semantically contracted if and only if $\mathcal{P}'$ does not match a single unique path Q_u. Multiple identified unique paths Q_u are normalised where different path fragments in all Q_u are replaced with '//'. Same path fragment in all Q_u are retained in $\mathcal{P}'$. This same path fragment has only some parent-child '/' between the pair of elements.*

Given a valid XML schema graph $\mathcal{G}$, we derive a set of unique paths $\mathcal{Q} = \{Q_{u1}, Q_{u2}, Q_{u3}, \ldots, Q_{nn}\}$. Each Q_u has a query tree pattern ($\mathcal{V}_Q$, $\mathcal{E}_Q$) where $\mathcal{E}_Q$ is a set of edges '/' and set of elements $\mathcal{V}_Q$ in each Q_u.

The "iff" direction in the Proposition indicates that an XPath query $\mathcal{P}$ has a set of elements and edges where elements are labelled with l or '' or '.' and edges are expressed by '|' or '||'. If all elements in some Q_u are a superset of all elements in $\mathcal{P}$, and '*' or '.' in $\mathcal{P}$ can be mapped to elements in same set of Q_u and edge trees '||' (if there exists any) cannot be replaced by a sequence of elements labelled l separated by '/' in Q_u then set Q_u some different path fragments and same path fragment. The pattern normalisation would produce $\mathcal{P}'$ that has a sequence of elements separated only by '/' (repeated path fragment in set Q_u and '//'(different path fragment in set Q_u).*

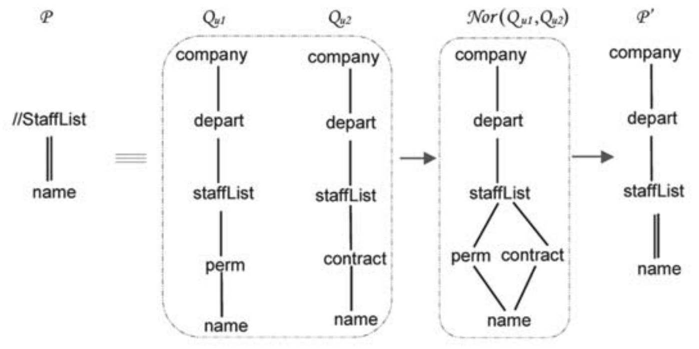
Rule 2 (Semantic path contraction)**:** *XPath query $\mathcal{P}$ is semantically contracted if it meets the following requirements*:

- *Seek the target element of $\mathcal{P}$ to target element in each unique path Q_u to identify the number of possible Q_u.*
- *For all unmatched elements in $\mathcal{P}$, match each of them to those in identified possible Q_u to ensure that tag name is conflict free.*
- *Detect "//" in $\mathcal{P}$ and match it back to a path fragment in each identified Q_u, '//'.*
- *Detect all '..', '*' and '.' in $\mathcal{P}$ and replace them with matched elements in each Q_u.*
- *Normalise all identified Q_u where the repeated path fragment in all Q_u is retained and the different path fragments in all Q_u are contracted to '//'.*
- *$\mathcal{P}'$ is the semantic contracted XPath query that is derived in the form of set of valid elements separate by '/' and '//'.*

Let us consider an example of XPath query $\mathcal{P}$ = //staffList//name. (2)

Using the schema information from Figure 1, Figure 4 depicts $\mathcal{P}$, set of identified Q_u and $\mathcal{P}'$ in tree patterns.

Figure 4 $\mathcal{P}$, Q_u & $\mathcal{P}'$



$\mathcal{P}$ = //staffList//name $Q = \{Q_{u1}, Q_{u2}\}$
$\mathcal{V}_{\mathcal{P}} = \{$staffList,name$\}$ Q_{u1} = company/depart/staffList/perm/name
$\mathcal{E}_{\mathcal{P}} = \{//\}$ Q_{u2} = company/depart/staffList/contract /name
$\mathcal{V}_{Qu} = \{$company, depart, staffList, perm, contract,name$\}$
$\mathcal{E}_{Qu} = \{/\}$ where $i = \{1, 2\}$

$\mathcal{P}'$ = company/depart/staffList//name

By applying *Rule 2* to $\mathcal{P}$, the target element 'name' $\in \mathcal{V}_{\mathcal{P}} \rightarrow$ 'name' $\in \mathcal{V}_{Qu}$ where $i = 2$, element 'staffList' in $\mathcal{P}$ can also be mapped to element 'staffList' in both Q_{u1} and Q_{u2}. Edge '//' is detected between 'staffList' and 'name' elements in $\mathcal{P}$ means the target element 'name' is traversing back to element 'staffList' on two different path fragments 'staffList/perm/name' and 'staffList/contract/name' respectively. The difference in path fragment from 'statffList' to 'name' appears Q_{u1} and Q_{u2}. Figure 4 depicts a query pattern during a normalisation of Q_{u1} and Q_{u2} denoted $\mathcal{N}or^{4}(Q_{u1}, Q_{u2})$ where 'company/depart/staffList' are the repeating nodes or path fragments in both Q_{u1} and Q_{u2} from the root to the 'staffList' element. According to the semantic path contraction rule, we would replace the different fragments in Q_{u1} and Q_{u2} with '||' as shown in the $\mathcal{P}'$ tree pattern in Figure 4.

As the example demonstrates, the *semantic path contraction* is applied to an XPath query only when it matches multiple unique paths after undergoing the pattern matching process.

4.3 Semantic path complement transformation

Without the use of a predicate [ ] in an XPath query, it can still have a condition represented by '..'. Although the '..' operator. Due to the users' nature, they may choose to use '..' instead of [ ]. Therefore, our semantic path transformation provides a complete solution to transform such an XPath query. The proposed semantic complement transformation allows us to complement a new XPath query by collaborating with both the semantic path contraction and semantic path expansion. In doing so, we can eliminate the '..' based on the occurrence constraint of the conditional[5] element; or we could transform '..' to [ ] if elimination is not applicable.

Definition 13 (Semantic path complement)**:** A semantic path complement $\mathcal{P}'$ is a transformation from the XPath query $\mathcal{P}$ where $\mathcal{P}'$ is a unique path that may or may not contain a condition enclosed by [ ].

Proposition: *$\mathcal{P}'$ is a semantic complementary of $\mathcal{P}$ if there exists branching elements (s) (..) in $\mathcal{P}$ and $\mathcal{P}'$ may be enclosed with predicate [ ] if a conditional element cannot be eliminated.*

Given a valid XML schema graph $\mathcal{G}$, we derive a set of unique paths $Q = \{Q_{u1}, Q_{u2}, Q_{u3}, \ldots Q_{nn}\}$. Each Q_u has a query tree pattern ($\mathcal{V}_Q$, $\mathcal{E}_Q$) where $\mathcal{E}_Q$ is a set of edges '/' and set of elements $\mathcal{V}_Q$ in each Q_u.

The "iff" direction in the Proposition is for a given XPath query that has a set of elements labelled with l,'', '.', '..' and edges are '|' or '||'. All elements in some unique paths Q_u are a superset of all element in $\mathcal{P}$; '..' indicates an existence of parent element in $\mathcal{P}$. The edge trees '||' in query tree pattern $\mathcal{P}$ can be replaced by a sequence of elements labelled by l and edge '|'. The 'iff' direction also ensures that '..' and conditional element are eligible to be removed from $\mathcal{P}$ when the parent of the target element is also the parent of the conditional element and its occurrence (or multiplicity) constraint must be greater than or equal to 1. In some classic cases, both '..' and conditional element cannot be removed as the occurrence constraint of the conditional element equals to 0, then '..' is replaced and the conditional element is enclosed by [ ].*

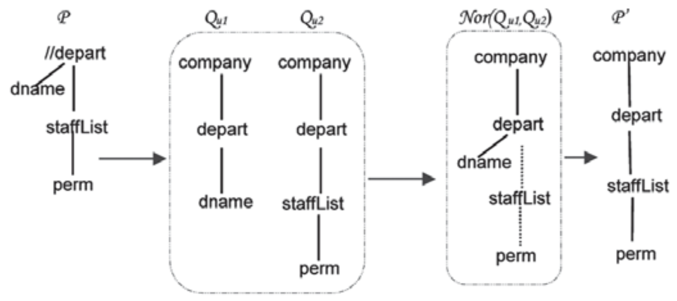
Rule 3 (Semantic path complement)**:** *XPath query P is semantically complemented by the following requirement criteria*:

- *Seek the target element of $\mathcal{P}$ to target element in each unique path Q_u to identify the number of possible Q_u.*
- *For all unmatched elements in $\mathcal{P}$, match each of them to those in identified possible Q_u to ensure that the tag name is conflict free.*
- *Detect '..' in $\mathcal{P}$ and determine the number of branching nodes based on the m occurrences of '..' in $\mathcal{P}$. Each path expression that traverses from the root of the schema to each child element (used by a predicate or for a conditional requirement) of the branching element must match with one unique patch Q_u, otherwise a conflict in structural semantic will be detected.*
- *Normalise all identified Q_u where the repeating path fragment in all Q_u is retained. The conditional element is removed only when its occurrence is 1 or above.*
- *There may exist [ ] in semantic path $\mathcal{P}'$ iff the minimal occurrence of a conditional element is 0.*

Let us consider an example of XPath query $\mathcal{P}$ = //depart/dname/../staffList/perm. (3)

Using the schema information from Figure 1, Figure 5 shows $\mathcal{P}$, Q_u and $\mathcal{P}'$ expressed in tree patterns.

Figure 5 $\mathcal{P}$, Q_u, $\mathcal{N}or(Q_{u1}, Q_{u2})$ & $\mathcal{P}'$



$\mathcal{P}$ =//depart/dname/../staffList/perm

$\mathcal{V}_{\mathcal{P}}$ = {depart,dname,staffList,perm,..}

$\mathcal{E}_{\mathcal{P}}$ ={'/', '//'}

$\mathcal{P}'$ = company/depart/staffList/perm

$Q = \{Q_{u1}, Q_{u2}\}$

Q_{u1} = company/depart/dname

Q_{u2} = company/depart/staffList/perm

$\mathcal{V}_{Qu}$ = {company,depart,dname,staffList,perm}

$\mathcal{E}_{Qu}$ = {'/'} where $i = \{1, 2\}$

By applying *Rule 3*, the query tree pattern depicted in Figure 5, '..' is detected and as children under the branching node 'depart' is 'dname' and 'staffList' in $\mathcal{P}$. Two unique paths Q_u 'company/depart/dname' and 'company/depart/staffList/perm' are associated with a path expression from 'company' element to 'dname' and a path expression from 'company' element to 'perm' element. By applying $\mathcal{N}or(Q_{u1}, Q_{u2})$ with the occurrence constraint of 'dname' element (based on XML schema information in Figure 1) is 1, 'dname' can be removed from 'company/depart/dname' which left 'company/depart'. However, 'company/depart' is the subset of unique path 'company/depart/staffList/perm'. This means $\mathcal{P}'$ is the unique path 'company/depart/staffList/perm'.

This illustrates a transformed XPath query $\mathcal{P}'$ without [ ]. Our running example 3 demonstrates the use of the *semantic path complement rule*.

4.4 Semantic qualifier transformation

Semantic qualifier (*predicate*) *transformation* is the transformation dealing with a predicate enclosed by [ ]. An element used for conditioning in a predicate [ ] can be a leaf-node or element type. In the case where the element is a leaf-node, it may or may not have values. Acceptance values defined for a leaf-node, rather than atomic type, in the schema are known as constraint values or acceptance content. In the case where an element or a fragment of XPath being enclosed by the predicate [ ] is a type, it shows no values for conditioning, which normally relies on the last element in the path fragment. Another exceptional scenario is where there is only a value or a context sensitive (Berglund et al., 2007) function such as last() or position() with a value enclosed in [ ], this is known as *index ordinary* value on the current element.

Let us consider or an example of XPath query 'company/deparment[//perm [position()>=1]]/dname. To transform this type of query, we first look at the condition in the predicate whether the condition should be left of using '//' or possibly to expand it to a full conditional path such as 'staffList/perm'. To achieve this type of transformation, we first employ the semantic path expansion transformation. The next constraint we also need is the occurrence constraint of 'contract', which determines whether there is an occurrence constraint between ‘dname’ and ‘contract’ employee. Suppose we apply some transformation rules, which can produce an equivalent query such as ‘company/depart/dname’.

The question we face is which one of these constraints to use first. Conflict constraint will produce an empty returned result set, but most importantly, if the query is satisfied, would the decision on processing steps of transformation provide an opportunity for an increased performance?

Before we provide the transformations, we discuss the terminologies that support our *semantic qualifier transformations.*

Definition 14 (Full-qualifier)**:** A full-qualifier is a predicate [ ] of which all values of the elements enclosed by [ ] must be equivalent to all the values of the same element defined in the schema.

For instance, a predicate [location = ‘Bundoora’ or location ‘Melbourne’] appears in an XPath. Referring to the schema definition, an element ‘location’ is defined with two restricted values {‘Bundoora’, ‘Melbourne’}, and in this case the predicate satisfies a full-qualifier. However, a predicate [location = ‘Bundoora’ and location ‘Melbourne’] or a predicate [location = ‘Bundoora’] is not a full-qualifier due to the AND used in the first case and restricted value is only partially validated for the ‘Bundoora’ value in the second case, which leads to the next definition.

Definition 15 (Partial-qualifier)**:** A partial-qualifier is a predicate [ ] of which all values of the element enclosed by predicated [ ] are equivalent or matched to some but not all values of the same element defined in the schema.

For instance, a predicate [location = ‘Bundoora’] appears in an XPath. Referring to the schema definition, an element ‘location’ is defined with two restricted values “Bundoora”, “Melbourne”; in this case the predicate satisfies the condition of being a partial-qualifier. However, a predicate [location = ‘Bundoora’ and location ‘Melbourne’] is not a semi-qualifier.

We now introduce a set of formal notations accompanied by a brief description of each as these will be used repeatedly throughout the definitions and rules being presented in this section:

- $\mathcal{F}n$ *is a series of context functions such as last(), position(), etc…*
- $\mathcal{O}p$ *is a series of comparative operators* {!=,=, <, >,>=, <=}
- θ *is the restricted values of an element*
- $\mathcal{N}$ *is a series of index values, restricted element with or without a restricted value, or restricted path fragment. In general* $\mathcal{N}$ *is referred to as an inner focus of* [ ].

In this section, we introduce two semantic qualifier transformations:

- *Semantic ordinal index transformation* allows the transformation of an XPath query in the presence of the *ordinal index predicate*. We utilise the Occurrence constraint of the element for which the index value is l and greater than 1.

- *Semantic Content-based Transformation* allows the transformation of an XPath query in the presence of values of a conditional element specified in the predicate. This has been addressed earlier in the work of Su et al. (2004, 2005) in the semantic optimisation context. The difference between our work and earlier works is that we introduce more constraints on the leaf-node and combine this with the semantic path transformation to provide a complete transformation.

A Semantic ordinal index transformation

We apply the *multiplicity* (also known as *occurrence*) constraint to transform the *ordinal index* predicate. To the best of our knowledge, none of the earlier works has addressed this type of predicate.

The function *position*() is used to return the context position. The context position is the position of the context item within the sequence of items currently being processed. When this function is used with a value in the predicate, it returns the context node or item based on a given position value. When the *position*() function is used without a given value in the predicate, all nodes of the current item being processed are returned. A predicate may contain only an integer value, which denotes the position value of the current element. For example, a query 'company/depart[position() = 1] is equivalent to 'company/depart[1]'.

The function *last*() is used to return the context size. The context size is the number of the items or context nodes in the sequence of items being processed. When the *last*() function is used without a given value in the predicate, it takes the default value as the last item in the sequence returned.

In an ordinal index predicate, if a *fn* such as *position*() or *last*() is an operator o_p w.r.t $o_p \in O_{\mathcal{P}}$, a value θ or a *fn* is expected; if one of these is missing, then a semantic conflict is immediately detected.

Definition 16 (Semantic ordinal index predicate)**:** A semantic Ordinal Index predicate is a transformed predicate that may or may not have an enclosed [ ] in a semantic XPath query $\mathcal{P}'$.

Proposition: *$\mathcal{P}'$ is a valid transformed XPath query where the ordinal index predicate possibly can be eliminated or retained iff the predicate is a full-qualifier or partial-qualifier respectively.*

Given a valid XML schema graph $\mathcal{G}$, we derive a set of unique paths $\mathcal{Q} = \{Q_{u1}, Q_{u2}, Q_{u3}, \ldots, Q_{un}\}$. Each Q_u has a query tree pattern formed by a set of elements $\mathcal{V}_Q$ and edges $\mathcal{E}_Q$ where $\mathcal{E}_Q$ is a set of edges '/'.

The 'iff' direction in the proposition states that for a given query tree pattern of an XPath query that has a set of elements labelled with l, '', '.', '..' and edges '|' or '||'. For each valid element l, there may be a predicate [ ] which contains some context functions fn with or without a value following the operator o_p or index value with no function and operator used. The transformation $\mathcal{P}'$ would:*

- *eliminate the condition only when the current element being processed has an index value θ for the filtering such that $\theta = \sigma$; $\theta = \partial$; or $\theta = \sigma > \theta = \partial$ where $(\partial, \sigma) \rightarrow$ (minOccur, maxOccur) represent the minimal and maximal values of occurrence constraint on the current element such that $\partial = 1$, $\sigma = 1$; or $\partial = \sigma$ that. Both ∂ and σ are integers greater than* 0;

- *retain the predicate only when the current element being processed has an index value θ for filtering such that $\theta \leq \sigma$ or $\theta \geq \partial$ where* $(\partial, \sigma) \rightarrow$ (*minOccur, maxOccur*) *represent the minimal and maximal values of occurrence constraint on the current element such that* $\partial \geq 1$, $\sigma = \propto$; *or* $\partial \neq \sigma$. *Both* ∂ *and* σ *are greater than or equals* 0.

Rule 4 (Semantic ordinal index)**:** *An ordinal index predicate in an XPath query $\mathcal{P}$ is semantically transformed by the following necessary requirements:*

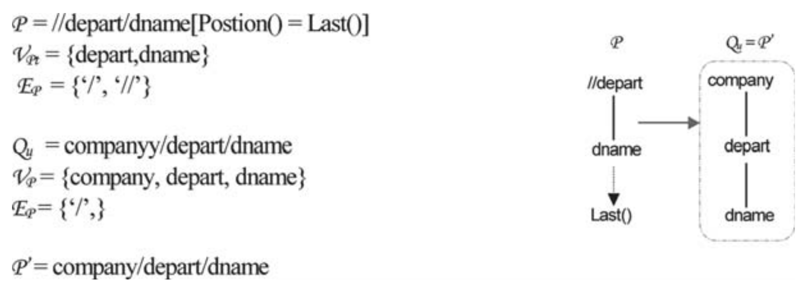
- *Detect [ ] in the path where the condition $\mathcal{N}$ expressed in the form of $\mathcal{N} = \exists fn \cup \exists o_p \cup \exists \theta$ or only θ.*
- *Identify a unique path Q_u that matches the structure of $\mathcal{P}$ excluding predicate [ ].*
- *For each condition in predicate [ ], verifying value against the constraint value of the branching element.*
- *Condition $\mathcal{N}$ is removed iff the value is 1 for position(),last(), position() = last() or self-index when the minimal and maximal values of the branching element are 1 as defined in the schema.*
- *Condition $\mathcal{N}$ is retained iff the value is not 1 for position(),last(), position() = last() or self-index when the minimal and maximal values of the branching element are not 1 as defined in the schema.*
- *Semantically expand, contract complement the whole path if possible.*

Let us consider XPath query $\mathcal{P}$ =//depart/dname[position()=last()]. (4)

When a context function is used in the predicate, the branching node in this case is the context node itself. Therefore, the content of predicate [ ] presented on the tree pattern is not applicable.

By applying Rule 4, as depicted in Figure 6, [ ] is detected in $\mathcal{P}$ where the inner focus is an ordinal index of $\mathcal{N}$ = [*position*() = *last*()]. The outer focus 'dname' is the valid node, but also a self-branching node as 'dname' is restricted itself, which is indicated by an arrow dot-line. Referring to Figure 1, there exists only one unique path Q_u = company/depart/dname, that leads to 'dname' from the root. Element 'dname' has a multiplicity or occurrence constraint of (*minOccur,maxOccur*) = (1,1); this allows [ ] to be removed from $\mathcal{P}$, that makes $\mathcal{P}'$ = company/depart/dname.

Figure 6 $\mathcal{P}$, Q_u & $\mathcal{P}'$


$\mathcal{P}$ = //depart/dname[Postion() = Last()]
$\mathcal{V}_{\mathcal{P}t}$ = {depart,dname}
$\mathcal{E}_{\mathcal{P}}$ = {'/', '//'}

Q_u = companyy/depart/dname
$\mathcal{V}_{\mathcal{P}}$ = {company, depart, dname}
$\mathcal{E}_{\mathcal{P}}$ = {'/',}

$\mathcal{P}'$ = company/depart/dname

We can also demonstrate the conflict of semantic/constraint in XPath query $\mathcal{P}$ = //depart/dname[position() = 2]. As the maximal occurrence of 'dname' in the schema is 1, it allows each ‘depart’ to carry only one name, so there would not be a second name for a ‘depart’ element in the XML database. This type of XPath query can immediately be determined by a semantic conflict.

B Semantic content-based transformation

We study specific constraints such as *enumeration, primary key, foreign key, pattern, inclusive and exclusive* for this content-based transformation. This semantic transformation adopts the transformed concepts such as elimination or introduction of range values, space or acceptance values proposed in a relational database. King (1981), Hammer and Jdondik (1980), Shenoy and Ozsoyoglu (1987) dealt mainly with predicates that contain record fields that are defined as simple or atomic data type. In a similar manner, we adopt them to deal with a transformation for predicates containing a leaf-node that could be an atomic element or attribute whose content is used for restriction.

First let us categorise the abovementioned constraints into two categories: *Space/Acceptance value* and *Range value,* and describe how each is used in our proposed techniques. The categorisation is:

1 *Space/Acceptance value*: Constraints such as enumeration, primary key, foreign key and pattern belong to this category, as validation of values requires the exact matching values. The eligible comparison operators would be ‘=’ or ‘!=’ and content-join[6] Π between the values.

  For example, refer to Figure 1 where ‘enumeration’ constraint on ‘location’ element is enumerated with “‘Bundoora’, ‘Melbourne’”, ‘Status’ is enumerated with “‘Active’, ‘Inactive’” and pattern restriction for ‘supervisor’ element is “[A–Z]{0–999}–[a–z]”.

2 *Range value*: Constraints such as inclusive and exclusive belong to this category. Validation of values used by elements in a predicate requires the range matching. The suitable comparison operators would be ‘>’, ‘>=’, ‘<’, or ‘<=’; content-Join Πwould also be used in this category but not compulsory.

  For example, referring to Figure 1, the ‘age’ element has a range value for employees that is greater than 22 and 55. For the range value restriction, it provides more flexibility on the expression of the condition. Element ‘age’ used by [ ] can be removed if it is a full-qualifier; this means that the content of the predicate [ ] under the various scenarios can be specified.

  Under this category, [ ] becomes a partial-qualifier if the join-content Π is used with the ‘=’ or ‘!=’ operator to perform the comparison. An out-of-range value used by an element in the predicate would result in a semantic conflict.

Let $[\mathcal{M}]$ is a content-based predicate; $\mathcal{M}$ is a set of condition, Π be a content-join; o_p be a comparison operator $o_p \in O_{\mathcal{P}}$ and $O_{\mathcal{P}} = \{!=, =, <, >, >=, <=\}$; l be a tag name of an element; θ be a value. $\mathcal{M} = n_1 \Pi n_2 \ldots \Pi n_2$ (Π and n_i can be null) *s.t.* $n_i = \exists(l|‘.’) \cup \exists o_p \cup \exists \theta$ where $(l|‘.’)$.

Rule 5 (Semantic content-based)**:** *Content based predicate in an XPath query $\mathcal{P}$ is semantically transformed in the order of the following steps*:

- *Detect* [ ] *in the path that has $\mathcal{N}$.*
- *Identify the unique path Q_u that matches $\mathcal{P}$; $\mathcal{P}$ may match as single unique path, if this is the case then the conditional element is the target element and condition in the predicate is the value of the target element. $\mathcal{P}$ may match a series of Q_u; if this is true then the conditional element in a predicate is the rightmost element in one of the associate unique path Q_u.*
- *For each n_i in $\mathcal{N}$, n_i is confirmed for a defined constraint value.*
- *$\mathcal{N}$ is removed iff $\mathcal{N}$ is a full-qualifier or $\mathcal{N}$ is retained if some n_i or all n_i of $\mathcal{N}$ are partial-qualifier.*

Let us consider an example of XPath query

$\mathcal{P}$ = //depart/location[. = 'Bundoora' or . = 'Melbourne']. (5)

We apply *Rule 5* where [. = 'Bundoora' or . = 'Melbourne'] is detected in $\mathcal{P}$. 'location' is a self-branching node. There is a matching Q_u = company/depart/location. A verification of self-branching node 'location' is a valid element in Q_u. For each n_i '.' is 'location' and its constraint is enumeration, belongs to acceptance value category. Since [. = 'Bundoora' or . = 'Melbourne'] has a content-join Π is 'OR' and all values of 'location' in [ ] matched all restricted values of 'location' defined in the schema, this confirms a full-qualifier for $\mathcal{N}$, hence it can be removed.

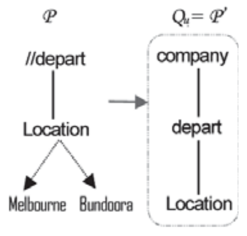
Figure 7 depicts the outer focus [ ] as a self-branching 'location' node and the content of predicate [ ] is the acceptance values. Reference to Figure 1, only one unique path Q_u Q_u = company/depart/location that traverse from 'company' to 'location'. We show the restriction values on the branching node in $\mathcal{P}$ but not Q_u in Figure 7.

Figure 7 $\mathcal{P}$, Q_u & $\mathcal{P}$'

$\mathcal{P}$= //depart/location[.='Bundoor'or.= 'Melbourne']
$\mathcal{V}_{\mathcal{P}}$ = {depart,location}
$\mathcal{E}_{\mathcal{P}}$ = {'/', '//'}

Q=company/depart/location
$\mathcal{V}_{\mathcal{P}}$ = {company, depart,location}
$\mathcal{E}_{\mathcal{P}}$= {'/',}

$\mathcal{P}$'= company/depart/location



Let us consider another example of XPath query

$\mathcal{P}$= //depart[location='Bandore' or location='Melbourne']/dname. (6)

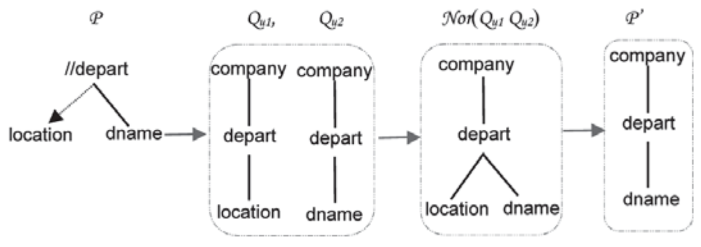
We apply *Rule 5* where [location = 'Bandore' or location = 'Melbourne'] is detected in $\mathcal{P}$. There are two matching unique paths Q_{u1} = company/depart/location, Q_{u2} = company/depart/dname. The normalisation process derives a normalised path where 'depart' is an element and also is the branching node in $\mathcal{P}$.

As shown in Figure 1, ‘location’ is the child of ‘depart’ and its constraint (motivation example) is *enumeration* that has acceptance-values {‘Bundoora’, ‘Melbourne’}. The predicate [location = ‘Bundoora’ or location = ‘Melbourne’] has a content-join Π ‘OR’ and all values of ‘location’ in predicate [ ] matched all values of ‘location’ in the schema. This confirms a full-qualifier for $\mathcal{N}$. The occurrence constraint defined in the schema for ‘location’ is (*minOccur, maxOccur*) = (1, 2) means that for each department there is at least one valid location, either ‘Bundoora’ or ‘Melbourne’. The occurrence constraint defined in the schema for ‘depart’ is (minOccur, maxOccur) has been left empty. By default, (*minOccur, maxOccur*) = (1,1) for any element if the element has no assigned value. Predicate [location = ‘Bundoora’ or location = ‘Melbourne’] can be eliminated based on the minimal occurrence value of ‘location’ under ‘depart’. Hence, $\mathcal{P}'$ = //depart/dname. We further apply semantic path expansion to optimise the transformation of $\mathcal{P}'$ = company/depart/dname.

Figure 8 depicts a query tree patterns of $\mathcal{P}$, Q_1, Q_2 and $\mathcal{P}'$. Notice the difference of tree pattern of $\mathcal{P}$ in Figures 7 and 8. We also show how the pattern is matched from $\mathcal{P}$ to $Q = (Q_1, Q_2)$ and Q to $\mathcal{P}'$. The normalisation process $\mathcal{N}or(Q_{u1}\ Q_{u2})$ produces $\mathcal{P}'$.

In this main section of our work, we propose a series of transformations ranging from path transformation to the predicate transformation in which a predicate is further addressed with different transformations to handle the ordinal index and content-based semantic predicates. Although we do not claim that this is a complete work of transformation, we do have to acknowledge that our transformations have the ability to deliver solutions for different types of XPath. By doing this, we are approaching the end of this whole search, which provides a complete, novel transformations technique using semantic information.

Figure 8 $\mathcal{P}$, Q_u, $\mathcal{N}or(Q_{u1}, Q_{u2})$& $\mathcal{P}'$



$\mathcal{P}$= //depart[location ='Bundoora'or location = ‘Melbourne’]/dname

$\mathcal{V}_{\mathcal{P}}$ = {depart, location, dname}

$\mathcal{E}_{\mathcal{P}}$ = {‘/’, ‘//’}

Q_{u1}= {Q_{u1}, Q_{u2}}

Q_{u1} =company/depart/location

Q_{u2} =company/depart/dname

$\mathcal{V}_{Q_u}$ = {company, depart, location,dname}

$\mathcal{E}_{\mathcal{P}}$= {‘/’,}

$\mathcal{P}'$= company/depart/dname

In the next section, we detail the two mains algorithms and describe the process of implementation including two phases. The first phase involves the pre-processing of the semantics in the schema and storing them so that they can be extracted and used by the transformation; the second phase is the transformation of an XPath query where the component is kept alive and ready to transform any input XPath query at any time.

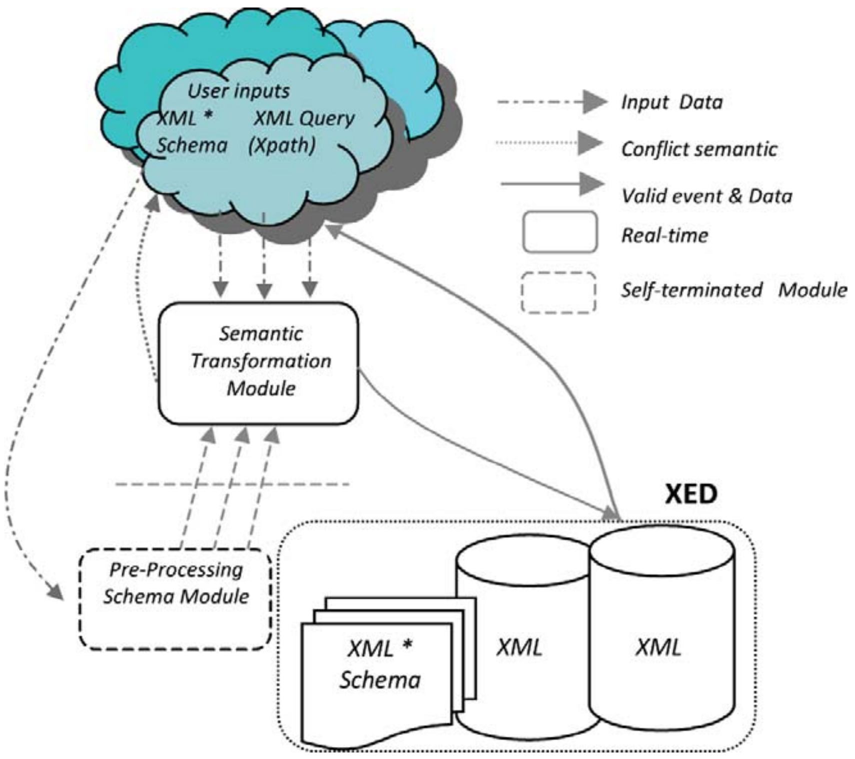
5 Implementation and evaluation

This section describes the detailed implementation including the framework overview, algorithms and evaluation of test results.

5.1 Overview of framework

The implementation overview of our semantic transformation is designed and shown in Figure 9.

Figure 9 Implementation overview of semantic transformation (see online version for colours)



On the start-up, the pre-processing schema (Le et al., 2007) system component is called first so that all the constraints/semantics defined in the given schemas are pre-processed. This module is shown by a dash-line rectangle labelled ‘Pre-processing schema module’. The processed semantics then are housed in the system memory; the pre-processing schema module is no longer needed it self-terminates (without human intervention).

The pre-processing schema component is restarted only if the schemas are changed/modified). The semantic transformation module takes over, which is invoked by the system if it detects a termination from the pre-processing schema module.

The semantic transformation module is a real-time module. Its functionality is to continue to accept the input XPath, to verify the constraints and transform the XPath if the constraints encounter no conflict. The semantic transformation sends the semantic XPath to the XED for performing the task. Otherwise, the XPath detects conflict in the semantic and a message is returned to inform the user. This whole semantic transformation module is shown as a thick-line rectangular box and is labelled ‘Semantic transformation module’.

Notice that in Figure 9, information enclosed by the dot rectangular box labelled ‘EXD’ is not part of our work. We use this same XML schema only for obtaining the constraints and related information to support our transformations. The XML schemas attached to the database and schema provided by the user must be the same, hence they are marked by ‘*’. The XML schema attached to the database simply means the data has been verified for consistency before the data is inserted into the database.

As we can see from the framework, our methodology will have a very minimal interaction with the database. In fact, the interaction with the database occurs only if the returned semantic XPath is a valid one. It proves that our transformation has an advanced ability to determine whether or not the query needs to access the database during the transformation phase.

5.2 Proposed algorithms

In Figure 9, we show two main tasks: task 1 is the pre-processing of schema and task 2 is the transformation of the XPath queries. The main program used to drive these two algorithms is excluded from this work. However, a brief description of the program functionality is to call the pre-processing schema module and semantic transformation module respectively.

The pre-processing schema algorithm is Algorithm 1. It accepts two input parameters including the XML schema and the name of the schema root. From these provided valid inputs, the schema is processed and three data structure lists $\mathcal{S}$, $\mathcal{C}$ and Υ are built respectively (Lines 5–7).

The first list $\mathcal{S}$ (Line 15) is a two-dimensional type to store the sequence elements in the form of [parent type][child] extracted from the schema. The very first item [root][$\mathcal{R}$] is inserted in $\mathcal{S}$ list (Line 9), the *etype* stores those detected as type (Lines 10–11). The subsequence items in the S list would be the root type and the children of root $\mathcal{R}$; those children are not the leaf-node.

The list $\mathcal{C}$ (Line 17) stores the semantics/constraints of all leaf-elements. The constraints are *cardinality, inclusive, exclusive, enumeration, primary key, foreign key, occurrence* and *pattern.* This list $\mathcal{C}$ is a three-dimensional type to store the semantic information in the form of [element name][constraint name][Val], where Val is a set of values, separated by a space, of a constraint. Otherwise, if a type is detected that is not a leaf-node, it is inserted into *eType* list (Line 18). The list Υ (Line 21) is the unique path list that stores all the derived unique paths in the form of a sequence of elements such as e.g., $e_1/e_2/e_3/e_n$, where e is the element, by calling on function list **unique_paths**.

The **unique_paths** accepts the $\mathcal{S}$, Υ lists and root name. It builds the unique paths by matching the last element of the current unique path to each item in $\mathcal{S}$ (until end of $\mathcal{S}$) where the parent type and child can be found. Once the child is found, it immediately builds the new unique path (Lines 1–10). Since efficiency is of concern when building this list of unique paths, the search for any given path from S list can be eliminated from the beginning if the given XPath query is a full path, in which case there is no need to perform the pattern matching.

Algorithm 1: Pre-processing Schema

```
1:  Input
2:      $\mathcal{T}$ = XSD schema
3:      $\mathcal{R}$ = root name of the schema
4:  Output
5:      $\mathcal{S}$ = List sequence elements defined in XSD Schema.
6:      $\Upsilon$ = List of unique paths derived from $\mathcal{S}$
7:      $\mathcal{C}$ = List of Leaf-node, constraint and values of constraints
8:  Begin
9:    insert root $\mathcal{R}$ into $\mathcal{S}$
10:   Let $eType$ = List of types.
11:   insert $\mathcal{R}$ into $eType$ where $\mathcal{R}$ is a type that has at least one child.
12:   while $eType$ not empty and not end of line ($\mathcal{T}$) read next line $\ell$
13:     if first element is a type $e_t$ from top of $eType$ list exist in $\ell$ then
14:       do read next line $\ell_n$ from schema start from line $\ell$
15:           insert $e_t$ and the element from $\ell_n$ into $\mathcal{S}$
16:           if the element has valid constraint and values
17:               insert the element, constraint and its values into list $\mathcal{C}$
18:           else if child is a type then insert type into $eType$
19:       until encounter $\ell$ contains no more $e_t$ in $\mathcal{T}$
20:     $\Upsilon$ = unique_paths ($\Upsilon, \mathcal{S},$ )
21: return $\mathcal{S}, \mathcal{C}, \Upsilon$

Function list unique_paths ($\Upsilon, \mathcal{S}, \mathcal{R}$)
1:  insert $\mathcal{R}$ into $\Upsilon$ where $\mathcal{R}$ is the root element
2:  set current unique path $u$ is $\mathcal{R}$
3:  while not end of $\Upsilon$
4:      Let $temp_u$ be a temporary unique path where $temp_{u}$:=NULL
5:      do each item in $\mathcal{S}$ if found type of last element in $u$ then
6:          found new child $e$ make $temp_u$:= $u \cup '/' \cup e$
7:          insert $temp_u$ into $\Upsilon$
8:      until end of $\mathcal{S}$
9:      $u$: is next item in $\Upsilon$ after current unique path $u$
10: return $\Upsilon$
```

The pre-processing schema information can take some time to complete, but bear in mind that it is done only once and we can reuse the processed information over and over again.

The **semantic_transformation** algorithm (Algorithm 2) is called right after the "pre-processing schema" module is terminated. Information such as the three structural list $\mathcal{S}$, $\mathcal{C}$, and Υ is passed to this module by the pre-processing schema algorithm (Line 2).

Each XPath is received from the user, and the algorithm performs task. The appropriate function, based on the XPath query structure) is then called to accomplish each desired task (Lines 10–12).

Algorithm 2: Semantic Transformation

```
1: Input
2:    $\mathcal{S}, \mathcal{C}, \Upsilon$ = Lists passed in by Algorithm 1
3:    $\mathcal{Q}$ = User XPath
4: Output
6:  Begin
7:    let $\mathcal{P}$ = null be the result of transformation
8:    repeat
9:      let predicate be '[', $\mathcal{Q}$ be xpath input by user
10:       if $\mathcal{Q}$ is not null then
11:         if predicate in $\mathcal{Q}$ then call semantic_predicate_transform($\mathcal{Q}$, $\mathcal{P}$)
12:         else if no predicate in $\mathcal{Q}$ then call semantic_path_transform
13:   until user hits 'Esc'

FUNCTION semantic_predicate_transform($\mathcal{Q}$, $\mathcal{P}$)
1: Let $i$ be index-ordinal predicate, $v$ value-based predicate, $\chi$ structural predicate
2: let $p_c$ conditional element; $\mathcal{C}$ list of constraints, $c_c$ be constraints and values of
        element; min be minimal occurrence and max be maximal occurrence
3: for each predicate in $\mathcal{Q}$ do
4:   if predicate is $i$ and check $\mathcal{C}$ found $p_c$ of $c_c$ then
5:     remove $i$ from $\mathcal{Q}$ when $p_c$ has (min=max=[index]=[position()=last()])>0
6:     retain $i$ when $p_c$ has (min ≠ max ≠ [index] ≠ [position()≠ last()])≥ 0
7:     set $\mathcal{P}$ = 'conflict' if index is not within limit occurrences exit function
8:   if predicate is $i$ and check $\mathcal{C}$ list not found $c_c$ then check $\mathcal{S}$ if $c_c$ not found
9:      set $\mathcal{P}$ = 'conflict' then exit function
10: if predicate is $v$ where validate $\mathcal{C}$ found $c_c$ of $v$ then
11:   if $c_c$ is a range type then
12:      remove $v$ when value of $v$ within range of $c_c$ with AND/OR is considered
13:   if constraint is a space/acceptance value type then
14:      remove $v$ from $\mathcal{Q}$ if its values of matched with all value of $c_c$ (only
            'OR' join for same element or AND for different elements)
15:      retain $v$ if its values matches with only some values of $c_c$)
16:       set $\mathcal{P}$ = 'conflict' if $v$ contains 'AND' for same $c_c$ exit function
17: if (predicate is $v$ and validate not found $c_c$ in $\mathcal{C}$) or
      (predicate is $\chi$ and validate $\Upsilon, \mathcal{S}$ list not found $c_c$) then
18:   set $\mathcal{P}$ = 'conflict' exit function
19: if predicate is $\chi$ and validate $\Upsilon$ found $c_c$ is child/children of $p_c$
20:    remove $\chi$ from $\mathcal{Q}$ if $c_c$, has min-occur ≥1
21:    retain $\chi$ if $c_c$ has min-occur = 0
22: $\mathcal{P}$ = semantic_path_transform($\mathcal{Q}$, $\mathcal{P}$)

Function semantic_path_transform ($\mathcal{Q}$, $\mathcal{P}$)
1: let $\mathcal{J}$ = ['/','*','.','//','..'] be a set of path operators
2: if only $\mathcal{J}$[0] used to separate each pair of element in $\mathcal{Q}$ and $\mathcal{Q}$ is an exact match
     an $y_i$ in list $\Upsilon$ then $\mathcal{P}$ = $\mathcal{Q}$
3: else if only $\mathcal{J}$[0] used to separate each pair of element in $\mathcal{Q}$ and $\mathcal{Q}$ not match
          with any $y_i$ in list $\Upsilon$ then $\mathcal{P}$ = 'conflict'
4: else if detect $\mathcal{J}$[4] in $\mathcal{Q}$ then $\mathcal{P}$= semantic_path_complement ($\mathcal{Q}$, $\mathcal{P}$, $\mathcal{J}$)
5: else if detect $\mathcal{J}$[3] in $\mathcal{Q}$ then $\mathcal{P}$= process_path ($\mathcal{Q}$, $\mathcal{P}$, $\mathcal{J}$)

Function list propagate_path ($p_t$, $p_s$, $\mathcal{J}$, $e_j$, $e_m$)
1: let $temp_t$ = $p_t$, reset $p_t$ to empty
2: for each sub-path in $p_s$
3:   augment length of $p_t$ by $temp_t$.size
4:    for each path in $temp_t$
5:      replace part from $e_j$ to $e_m$ in current $temp_t$ with $p_s$ and insert $temp_t$ to $p_t$
6: return $p_t$

Function list sub_path ($e_j$, $e_m$, $p_s$, $\mathcal{J}$)
1: let $p$ = null be the temporary path, set $p_s$ to null
2: while find $e_j$ and $e_m$ in $\mathcal{S}$
3:   for each path from $e_j$ to $e_m$
4:     $p = e_j \cup \mathcal{J}[0] \cup e_k \ldots \cup e_m$
5:     insert $p$ into $p_s$
6:     set $p$ to null
7:   if $p_s$ is null then insert 'conflict' in $p_s$
8: return $p_s$

Function process_operators ($q$, $\mathcal{P}$, $\mathcal{J}$)
1: Let $p_t$, $p_s$ be list of temporary path, $e$ be target element
2: if $e$ in $q$ not match any $e$ in all unique paths in $\Upsilon$ then
3:     return $p_t$ ='conflict' exit function
4: else
5:   for each last $\mathcal{J}$[3] in $q$
6:     if $\mathcal{J}$[3] not leading $q$ and ($\mathcal{J}$[3] not preceded and tailed by any $\mathcal{J}$) then
7:        found $e_j$ as start element and $e_m$ as ended element
8:     else if $\mathcal{J}$[3] not leading $q$ and ($\mathcal{J}$[3] preceded by $\mathcal{J}$[1] and tailed by none $\mathcal{J}$) then
9:       for each element in $q$ shifts to left from $\mathcal{J}$[1] until no element in $\mathcal{J}$ matched
10:          found $e_j$ as start element and $e_m$ as ended element exit for
11:    else if $\mathcal{J}$[3] not leading $q$ and ($\mathcal{J}$[3] preceded and ended by $\mathcal{J}$[1]) then
12:      for each element in $q$ shifts to left from $\mathcal{J}$[1] until no element in $\mathcal{J}$ matched
13:           found $e_j$ as start element exit for
14:      for each element in $q$ shifts to right from $\mathcal{J}$[1] until no element in $\mathcal{J}$ matched
15:          found $e_m$ as ended element exit for
16:     else if $\mathcal{J}$[3] leading $q$ and ($\mathcal{J}$[3] ended by $\mathcal{J}$[1]) then
17:         $e_j$ is root element and set as start element exit for
18:      for each element in $q$ shifts to right from $\mathcal{J}$[1] until no element in $\mathcal{J}$ atched
19:          found $e_m$ as ended element exit for
20:      if $e_j$ and $e_m$ not null then seek $\mathcal{S}$ to find sub-path $p_s$ = sub_path($e_j$, $e_m$, $p_s$, $\mathcal{J}$)
21:      if $p_s$ not 'conflict' or $p_s$.length>0 then
22:         $p_t$ = propagate_path ($p_t$, $p_s$, $\mathcal{J}$, $e_j$, $e_m$) else exit function
23:     if $p_t$ not 'conflict' then
24:      for each path in $p_t$
25:         for each element in a path
26:            match element in $p_t$ to element in $\mathcal{S}$
27:            if $\mathcal{J}$[1] leading $p_t$ then replace $\mathcal{J}$[1] by root in $\mathcal{S}$
28:            else if $\mathcal{J}$[1] not leading $p_t$ then set all paths in $p_t$ to null exit for
29:            else if $\mathcal{J}$[2] exists $p_t$ then replace $\mathcal{J}$[2] by matched element in $\mathcal{S}$
30:            if $p_t$ is null then exit for
31:  if $p_t$.length = 1 then
32:    $\mathcal{P} = p_t$
33:  else if $p_t$.length = 1 then
34:    $\mathcal{P}$ = semantic_path_contraction ($p_t$, $\mathcal{P}$, $\mathcal{J}$)

Function semantic_path_complement ($\mathcal{Q}$, $\mathcal{P}$, $\mathcal{J}$)
1: let $\mathcal{Z}$ be a list of $q$ derived from $\mathcal{Q}$ where $n_q = (n-1)\mathcal{J}[4]$ and $n \geq 1$
2: let $\beta$ be the branching node in $\mathcal{Q}$
3: if found elements separate by $\exists\mathcal{J}$[1], $\exists\mathcal{J}$[2] and/or $\exists\mathcal{J}$[3] in $\exists q$ then
4:   for each $q$ in $\mathcal{Z}$ $q$ = process_operators($q_i$)
5:     if $q$ is 'conflict' then exit Function
6:     else insert $q$ back to $\mathcal{Z}$ to override the current $q$
7:  for each pair ($q_\beta$, $q_i$) in $\mathcal{Z}$ remove $q_\beta$
8:       if $e$ child of $\beta$ exist $q_\beta$ and $e$ has min-occur ≥ 1  $\mathcal{P} = \mathcal{P} \cup q_i$
9:       else $\mathcal{P} \cup q_i$[descendant element from $\beta$ in $q_\beta$]
10: return $\mathcal{P}$

Function semantic_path_contraction ($p_t$, $\mathcal{P}$, $\mathcal{J}$)
1:  for each element $e_t$ of the first path in $p_t$
2:     $Occurence_{et}$ = 1
3:     for each path after the first path in $p_t$
4:         if found $e_t$ in current path then increase $Occurence_{et}$ by 1
5:      if $Occurence_{et}$ == $p_t$.length() then $\mathcal{P} = \mathcal{P} \cup \mathcal{J}[3] \cup e_t$
6:  return $\mathcal{P}$
```

Assume that function **semantic_predicate_transform** is called first. This function accepts an XPath query $\mathcal{Q}$ & $\mathcal{P}$, which is the result of $\mathcal{Q}$ being transformed. If $\mathcal{Q}$ is detected with a constraint conflict, then $\mathcal{P}$ is encoded with an error message and no further transformation needed. As mentioned earlier in this paper, we propose the transformation of a predicate that is divided into two main categories.

Handling the first category for index ordinal predicate, which has an index value, with or without the integration of position() or last() function, of the self-branching node. The predicate [ ] is removed if the minimal, maximal occurrence, context functions and the index value (with or without the presence of context functions such as position ()/last()) are equivalent and much greater than 0 (Line 5). Otherwise, the index ordinal predicate is retained– not all are equivalent [Line 6]. Constraint conflicts occur if the index in the predicate is not within the occurrence range, or if the self-branching node is not valid (Lines 7–9).

Handling the second category for value-based predicate (Line 1), which has the leaf-node and restricted values inside [ ]. This particular predicate handles two types of restrictions: while the former is the constraint with acceptance/spaces values, the latter is the type that has range-restricted values. The range values predicate (Line 11–12) is rather simple compared with the rest of the other predicate types. The removal rule is applied only if the required information retrieved is between the minimal and maximal values (apply to inclusive or inclusive constraint) of the leaf-node. The join 'AND/OR' is valid as long as it has a join between the lower bound and the upper bound values. Space/acceptance values (Line 13) used for restriction on an element in [ ]. Predicate [ ] is qualified for being removed from Q if it is a 'full-qualifier'. If a constraint of an element in the predicate is defined with a series of values; the join required between the different values of the same element is 'OR'. The join 'AND/OR' between a pair of two different elements is also verified (Line 14).

Predicate [ ] is retained if some values of the element's constraint are not present in the [ ]; this includes join 'OR/AND' between the values of the same element or different values of different elements (Line 15). Constraint conflicts occur if the element in the predicate is not valid; that is, it does not exist in the $\mathcal{C}$ list; or the element is not the child of the branching node; or the join is 'AND' of the different values on the same element (Line 16). The validation is returned with a 'conflict' status when the predicate is detected as a content-based one and restricted elements (including values) do not exist in the schema (Lines 17–18).

As the transformation is no longer needed, the control is returned to the beginning to wait for the new XPath query. The predicate can be a structure type where the condition in predicate is a sequence of elements or just one element. The removal rule applies only when the inner focus is valid and the minimal value of the child element (if the predicate contains a sub-path, then the child element in this case is the first element) of the branching node must be at least one (Lines 19–21). $\mathcal{Q}$ can be further contracted, expanded or complemented (Line 22).

We now discuss the function **Semantic_path_transform** called by the **"semantic_transformation"** algorithm. It accepts an input XPath query $\mathcal{Q}$ and returns a semantic XPath query $\mathcal{P}$. Line 1 sets an array of path operators that are handled in the function. The function first checks if $\mathcal{Q}$ exists in the list of unique paths $\mathcal{Y}$ and assigns $\mathcal{Q}$ to $\mathcal{P}$ immediately and return $\mathcal{P}$ to the user; otherwise $\mathcal{P}$ is returned with a 'conflict' if $\mathcal{Q}$ does not exist in $\mathcal{Y}$. This applies only when $\mathcal{Q}$ appears to be in the expression of a sequence of elements which are separated by operator '/' (Lines 2–3). If operator '..' is detected (Line 4) then function semantic_path_complement is called on to complement $\mathcal{Q}$. Otherwise, the function process_path is called on to process other operators such as '//', '*' and '.', and to determine whether the path should be semantically expanded or contracted (Line 5).

If the function **process_path** is called on, it first accepts XPath queries q and a dynamic array of path operator $\mathcal{J}$. The returned semantic xpath $\mathcal{P}$ as the result of q has been processed. The function first detects the existence of the target element in q as well as in one or more unique paths in Υ. q is conflicted if the target element does not exist in list Υ (Lines 1–3). Otherwise, the function backtracks from the target node in q processing all operator '//' (Lines 5–19). If '//' in q is valid, then it can be replaced by a sequence set of element in $\mathcal{S}$. The validity of '//' is determined by two valid elements that lead and end '//'. This pair of elements is then passed on to the sub-path function (Line 20) for deriving the sub-paths from list $\mathcal{S}$ to replace '//' by calling the propagate_path (Line 22). Finally, the operator such as '*' or '.' is individually processed in each p_t (Lines 23–30). Finally, each p_t in the list should contain only a sequence of elements in which each pair is separated by '/'. When p_t list contains only one p_t then q has semantically expanded (Lines 31–32) to p_t which is also $\mathcal{P}$. If there is more than one p_t in the list, then $\mathcal{P}$ is semantically contracted (Lines 33–34) from the list p_t.

The **propagate_path** function propagates p_s to '//' in q and returns a set of XPath query p_t based on the current number of p_s. which results from '//'. There may be more than one path fragments that traverse from branching and descending elements of '//' (Lines 1–6).

Function **sub-path** processes list $\mathcal{S}$ based on the pair of elements that leads and ends current '//', and returns a list type p_s. This list contains a sub-path or multiple sub-paths (Lines 1–8).

Function **semantic_path_contraction** derives a contracted path from p_t list; from the target element of all p_t traversing back to the root, which is also the branching node of each p_t. The function normalises the common elements in all p_t and replace those are not matched '//'. The contracted path $\mathcal{P}$ contains a sequence of elements separated by '/' as long as each element being found in all processed paths and replace the branches by '//' (Lines 1–6).

Function **semantic_path_complement** performs a number of XPath queries q from user input XPath query $\mathcal{Q}$. For each q, it is stored in list $\mathcal{Z}$ (Line 1). The branching node $\mathscr{b}$ (Line 2) is also identified. When any q in Z is detected with operators other than '/', it is processed further by calling on the process_operator (Lines 3–4) function. The returned expression q is either a valid path that contains only elements which are separated by operator '/' and/or '//' only; no other operators should exist. If the elements in q are not matched elements as defined in the schema (Lines 5–6), the whole transformation for $\mathcal{Q}$ is no longer valid, and the control returns to waiting state for a new $\mathcal{Q}$. Removal applies when the conditional element has a minimal occurrence whose value is 1 and above; the semantic complemented path $\mathcal{P}$ is q left in $\mathcal{Z}$ excluding restricted q (Lines 7–8). When restricted q cannot be removed; the '..' operator is replaced by [ ] and inner focus would be the sequence of descendants of the branching node $\mathscr{b}$ (Line 9). Line 10 returns the semantic complemented path $\mathcal{P}$ to the main control for accessing the database.

6 Empirical performance evaluation

The above algorithm has been implemented and a test bed has been built. We now evaluate the potential for optimisation produced by the proposed *Semantic transformations* using the results collected in the test bed system.

6.1 Performance evaluation strategy and experimental set-up

There exist essentially two practical approaches to the management of XML data and to the processing of queries to XML data: a Native XML database system and an XML-enabled database system. We therefore compare the performance of queries and transformed queries in our workload for two representative systems: a Native XML Database system (referred to as XMS) and a commercial XML Enabled Database system (referred to as XDB). The analysis of the results is twofold:

1 an evaluation ascertains the potential for optimisation (whether a transformed query can be evaluated more efficiently)

2 a comparison is carried out on the relative performance of the two systems.

XMLSpy[7] data generator generates XML documents conforming to a company.xsd schema as depicted in Figure 1. We use six data sets (compliant with this schema) of varying sizes: 15, 20, 25, 30, 35 and 40 mega bytes.

Table 2 consists of the queries and their transformed equivalent queries. The time response measurements are made using the profiling tools of the respective systems and are averaged (outer layers are eliminated) over several attempts in order to cater for possible interferences from the operating system and other system processes. The experiments are performed on a PC AMD64 (989) 3200+ with 2.0 GB of RAM disconnected from the network.

Table 2 XPath and Semantic Counterparts

No.	*XPath queries* (*Q*)	*Semantic XPath queries* (*SQ*)
1	//depart/staffList/*/email	company/depart/staffList//email
2	/*//perm/ages	company/depart/staffList/perm/ages
3	/*/depart/*/perm	company/depart/staffList/perm
4	//@location	company/depart/@location
5	//depart/staffList/perm/../contract	company/depart/staffList/contract
6	/*/depart/staffList//name	company/deparment/staffList//name
7	company/depart/staffList/perm[name/lastname]	company/depart/staffList/perm
8	company/depart/staffList/contract [city = "Bundoora" or city = “Altona”]	company/depart/staffList/contract [city = ‘Bundoor’]
9	company/depart/staffList[position()>=1] /perm/name[lastname=‘Smith’]	company/depart/staffList/perm/name [lastname=’Smith’]
10	company/depart/staffList/contract [ages > 22 and ages < 56]/name	company/depart/staffList/contract/na me

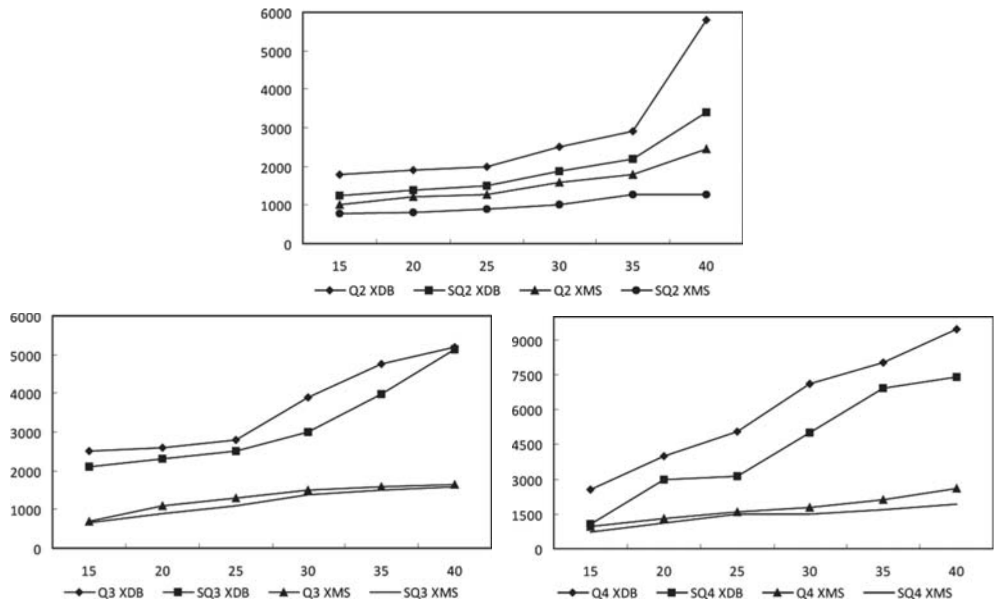
6.2 Results and analyses

The XPath queries are grouped and studied under each proposed semantic transformation.

A reminder here that XDB means XML-Enabled Relational Database and XMS means XML Native Database.

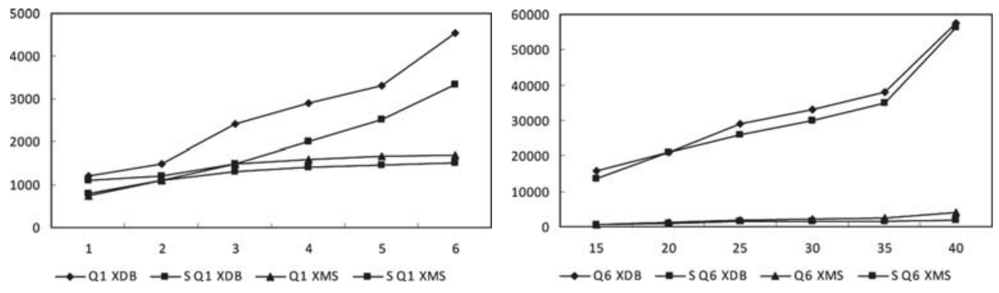
i **Semantic Path Expansion Transformation** for XPath queries *Q*2, *Q*3 and *Q*4, which refer to Table 2 and Figure 10 for result evaluation.

Figure 10 Semantic path expansion transformation



XPath query Q2 is specified with elements, a wildcard selection ‘*’, a ‘//’ ancestor-descendant and ‘/’ parent-child operators. XPath query (SQ2) is semantically expanded from Q2, which includes only a set of valid elements and ‘/’. Referring to Figure 11, the result shows a significant improvement in response time by about 90% SQ2 in XDB database. In XMS database, SQ2 shows a sound reduction of more than 50% in response time. This significantly improved performance is mainly due to the use of ‘//’ leading the XPath query, as the ‘*’ operator represents a valid element root ‘company’, which does not contribute much to this improvement.

Figure 11 Semantic path contraction transformation



XPath query *Q*3 is specified with only a set of valid elements, '/' parent-child operators and '*' operators. In this XPath query, each '*' operator (refer to Figure 1), represents 'company' and 'staffList' respectively. Similar to the use of '*' in Xpath query *Q*2, we can see the performance of Xpath query SQ3 in both XDB and XMS databases is improved by about 5%; it shows a very minor improvement in the transformed XPath queries. Operator '*' does not contribute much to the deterioration of performance.

On the other hand, XPath query Q4 demonstrates the use of an ID and '//' operator leading the XPath query. After undergoing the transformation process, the performance of SQ4 shows a significant improvement by about 90% in XDB database and sound reduction of response time by about 8% in the XMS database.

ii **Semantic Path Contraction Transformation** for XPath queries *Q*1 and *Q*6, which refer to Table 2 and Figure 11 for result evaluation.

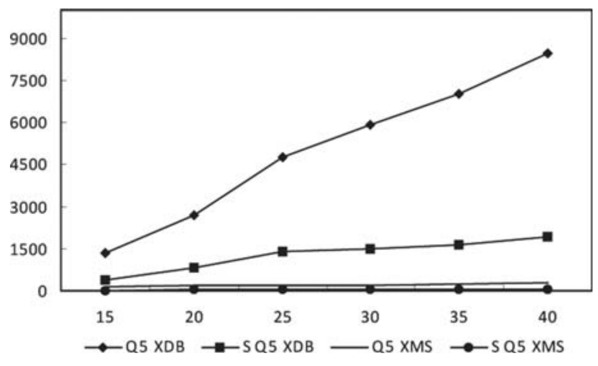
Apart from the use of '/' and valid elements, XPath query *Q*1 is also specified with '//' ancestor-descendant operator leading the XPath query, '*' unidentified element operator. After undergoing the transformation process, '//' can be expanded from 'depart' to root 'company'. However, '*' represents 'perm' and 'contract' elements. This shows that there are multiple unique paths that traverse from the root to 'email' via 'staffList' in order to produce the same result as of *Q*1. By applying our proposed transformation for contraction, as a result, the transformation has replaced '/*/' with '//'. The performance of SQ1 in XDB database shows a significant improvement by about 95% and less than 9% for the SQ2 in XMS database.

In XPath query *Q*6, which is specified with '*' as the leading element in the XPath query and ancestor-descendant '//'. After undergoing the transformation process, '*' is replaced with a valid element 'company' and the rest remain the same. As a result, the SQ6 in both XDB and XMS databases shows only minor improvement; this is due to the contribution of '*' in the original query.

iii **Semantic Path Complement** for XPath query *Q*5, which refer to Table 2 and Figure 12 for result evaluation.

XPath query Q5 is specified with '//' ancestor-descendant, '/' parent-child and '..' parent and set of valid elements. After going through our semantic path complement transformation process, in XPath query SQ5, '..' has been removed and '//' is replaced with a root 'company' element. There is a significant optimisation by about 95% for SQ5 in XDB and a slight improvement for SQ5 in XMS.

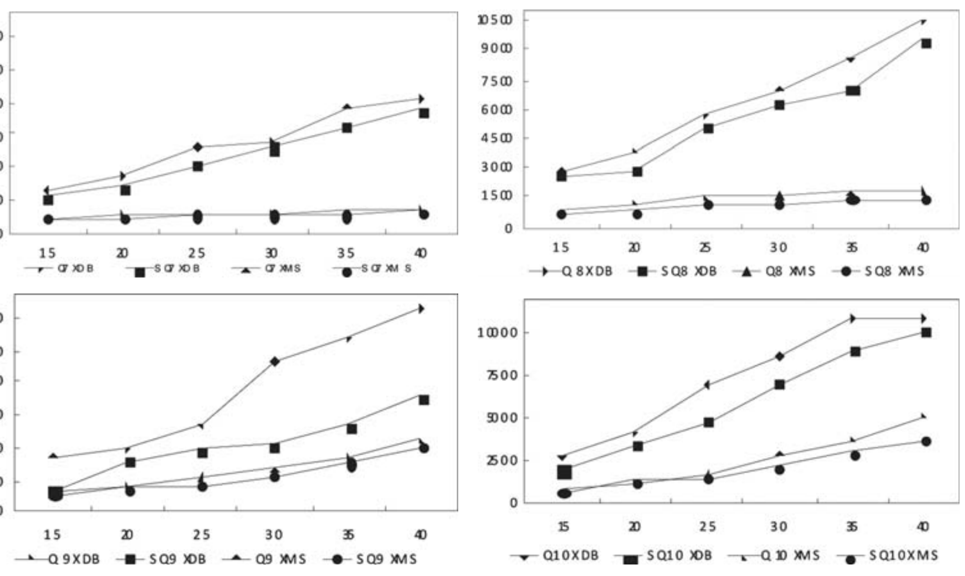
Figure 12 Semantic path complement



iv **Semantic Qualifier Transformation** (Semantic Ordinal Index transformation and semantic content-based transformation) for XPath queries *Q*7, *Q*8, *Q*9 and *Q*10, which refer to Table 2 and Figure 13.

The XPath queries *Q*7 and *Q*10 demonstrate the content-based transformation where predicates are restricted by range values (ages are between 22 and 55) and acceptance values (city is Bundoora or Melbourne}.

Figure 13 Semantic qualifier transformation



In addition to this, we also demonstrate the use of conjunctive join in the predicate in *Q*10. Both XPath queries *Q*7 and *Q*10 are successfully transformed. As a result, the predicates in these XPath queries depict a full-qualifier. The predicates are therefore eligible for elimination. As a result, the SQ7 and SQ10 in XDB database show a great improvement in performance of between 50% and 78%; however, the performance of these semantic XPath queries in the XMS database shows very little improvement.

In XPath query *Q*8, we demonstrates the use content-join of values “city = ‘Bundoora’ or ‘Altona’”. Even though ‘Altona’ is not an acceptance value defined for City in the schema, however, the join-content for values of ‘City’ is an OR operator. The predicate [city = ‘Bundoora’ or city = ‘Altona’] is identified as a partial-qualifier as “Altona” is regarded useless and can be removed. We found that there is a significant improvement by about 50% for SQ8 in XDB and about 18% for SQ8 in XMS. The result indicates that there is a great optimisation opportunity for the removal of values in the predicate.

XPath query *Q*9, we demonstrate the use of the context-sensitive node function and a join content-based. The first predicate uses the context function [position()>=1] on ‘staffList’ element which can be easily removed as, according to the occurrence ‘staffList’ set in the schema, there should be at least 1 staffList or multiple staffList in the database. This predicate [position()>=1] is identified as a full-qualifier. The second predicate uses [lastname = ‘Smith’] on the ‘name’ element. This predicate is identified as a partial-qualifier as there maybe other ‘lastname’ in the database. After going through the transformation process, [position()>=1] predicate is removed and [lastname = ‘Smith’] is retained.

Performance wise, there is not much improvement for SQ9 in XMS database with the small data size; nevertheless, the improvement picks up as data size increases. However, there is a huge improvement for SQ9 in XDB database (not only with the small data size) and this increases in big portions as the database size increases.

7 Conclusion and future work

We have proposed a family of semantic transformations of XPath queries including semantic path transformation and semantic qualifier transformation. The semantic path transformation is further divided into semantic path expansion, semantic path contraction and semantic path complement. The semantic qualifier transformation is further divided into two types of transformations. The former is the semantic ordinal index and the semantic content-based. Our semantic transformations utilise XML schema constraints as the main source.

The semantic path transformation is for transforming XPath queries that strictly exclude predicates. We focus on eliminating the use of operators such as ‘..’, ‘.’, ‘//’, ‘*’ as much as possible. Hence, we replace those with valid full XPath query or as many valid elements as we can.

The qualifier transformation is for transforming predicates in XPath queries. For this particular typology, we apply the elimination concept to the whole predicate or part of the predicate.

For completeness, we also devised a set of algorithms and implemented them for evaluation. Upon the completion of our implementation, we quantified empirically the potential of these transformations in two systems representative of the existing options for the management of XML data and for the processing of queries to XML databases. The results highlight potential opportunities in performance improvement, which although comparatively different, exist in both systems.

Our ongoing work focuses on fine tuning the algorithm and evaluating its cost (which is to be added to the response time in the case of ad hoc queries). Our preliminary observations suggest that pre-processing of the schema and appropriate data structures yields significant increase in performance even in the case of ad hoc queries.

References

Al-Khalifa, S., Jagadish, H.V., Koudas, N., Patel, J.M., Srivastava, D. and Yuqing, W. (2002) ‘Structural joins: a primitive for efficient XML query pattern matching’, *18th Proceeding of International Conference Data Engineering, IDCE*, San Jose, CA, USA, pp.141–152.

Amer-Yahia, S., Cho, S., Lakshmanan, L.V. and Srivastava, D. (2001) ‘Minimization of tree pattern queries’, *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, Santa Barbara, California, USA, pp.497–508.

Berglund, A., Boag, S., Chamberlin, D., Fernándezark, M., Kay, M., Robie, J. and Siméon, J. (2007) *Axes in XML Path Language (XPath 2.0)*, http://www.w3.org/TR/xpath20

Biron, P., Permanente, K. and Malhotra, A. (2004) *XML Schema Part 1 & 2*, http://www.w3.org/standards/techs/xmlschema#w3c_all

Charkravarthy, 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.

Che, D., Aberer, K. and Özsu, M.T. (2006) 'Query optimization in XML structured-document Databases', *The VLDB Journal – The International Journal on Very Large Data Bases*, Vol. 15, No. 3, pp.263–289.

Chen, D. and Chan, C. (2008) 'Minimization of tree pattern queries with constraints', *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, Vancouver, Canada, pp.609–622.

Coen, C.S., Marinelli, P. and Vitali, F. (2004) 'Schema path, a minimal extension to XML schema for conditional constraints', *Proceedings of the 13th International Conference on World Wide Web*, New York, NY, USA, pp.164–174.

Deutsch, A., Popa, L. and Tannen, V. (2006) 'Query reformulation with constraints', *SIGMOD Rec.*, New York, NY, USA, Vol. 35, No. 1, pp.65–73.

Groppe, S. and Groppe, J. (2006) 'A prototype of a schema-based XPath satisfiability tester', *Proceedings of 17th International Conference on Database and Expert Systems Applications*, Krakow, Poland, pp.93–103.

Gupta, K.A. and Suciu, D. (2003) 'Stream processing of XPath queries with predicates', *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, San Diego, California, USA, pp.419–430.

Hammer, M. and Jdondik, S.B. (1980) 'Knowledge-based query processing', *Proceedings of the 6th Very Large Databases (VLDB) Conference*, Montreal, Quebec, Canada, pp.137–146.

King, J. (1981) 'Quist: a system for semantic query optimization in relational databases', *Very Large Database (VLDB), IEEE Computer Society*, Cannes, France, pp.510–517.

Le, D. and Pardede, E. (2009) 'On using semantic transformation algorithms for XML safe update', *8th ISTA International Conference on Information Systems Technology and its Applications*, Sydney, Australia, pp.367–378.

Le, D., Bressan, S., Taniar, D. and Rahayu, W. (2007) 'Semantic XPath query transformation: opportunities and performance', *12th International Conference on Database Systems for Advanced Applications DASFAA*, Bangkok, Thailand, pp.994–1104.

Li, M., Mani, M. and Rundensteiner, E. (2008) 'ELF: a constraint-aware XQuery engine for processing XML streams with minimized memory footprint', *ICSC 2nd International Conference on Semantic Computing*, Santa Clara, CA, USA, pp.494–495.

Ramanan, P. (2002) 'Efficient algorithms for minimizing tree pattern queries', *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, pp.299–309.

Shenoy, S.T. and Ozsoyoglu, Z.M. (1987) 'Design and implementation of a semantic query optimizer', *IEEE Transactions on Knowledge and Data Engineering (1987)*, Vol. 1, No. 3, pp.344–361.

Smiljanic, M., Keulen, M. and Jonker, W. (2005) 'Formalizing the XML schema matching problem as a constraint optimization problem', *16th Conference on Database and Expert Systems Applications*, Copenhagen, Denmark, pp.333–342.

Su, H., Murali, M. and Rundensteiner, E. (2004) 'Semantic query optimization in an automata algebra combined XQuery engine over XML streams', *Proceedings of the 30th Very Large Data Bases (VLDB) Conference*, Toronto, Canada, pp.1293–1296.

Su, H., Rundensteiner, E. and Mani, M. (2005) 'Semantic query optimization for XQuery over XML streams', *Proceedings of the 31st Intl Conference on Very Large Data Bases*, Trondheim, Norway, pp.277–282.

Sun, W. and Liu, D. (2006) 'Using ontologies for semantic query optimization of XML databases', *Knowledge Discovery from XML Documents: First International Workshop on Knowledge Discovery from XML Documents (KDXD)*, Singapore, pp.64–73.

Wang, G., Liu, M. and Yu, J. (2003) 'Effective schema-based XML query optimization techniques', *Proceedings of the 7th Intl Database Engineering and Application Symposium, IDEAS*, Hong Kong, pp.1–6.

Wang, S., Su, H., Li, M., Wei, M., Yang, S., Ditto, D., Rundensteiner, E.A. and Mani, M. (2006) 'R-SOX: runtime semantic query optimization over XML streams', *Proceedings of the 32nd International Conference on Very Large Data Bases*, Seoul, Korea, pp.1207–1210.

Wood, P.T. (2001) 'Minimizing simple XPath expressions', *Proceedings of the Fourth International Workshop on the Web and Databases*, Santa Barbara, California, USA, pp.13–18.

Wood, P.T. (2003) 'Containment for XPath fragments under DTD constraints', *The 9th International Conference on Database Theory*, Paris, France, pp.300–314.

Notes

[1]Content-based predicate is a predicate that has a leaf-node and restricted value for data filtering. e.g., //employee[age > 22] where 'age' is an element in the predicate and 22 is the value of 'age'. Semantic content-based predicate is the transformation of content-based predicate, which will be defined later.

[2]Index ordinal predicate is *a predicate that contains only a value* (denoted as θ) *or a context sensitive* (Biron *et al.*, 2004) *function last*() or *position*(). Semantic Index ordinal predicate is the transformation of Index ordinal predicate, which will be defined later.

[3]The target node is the most important element in an XPath query. E.g., we have a unique path 'company/name' where element 'name' is the target node under the parent element 'company'.

[4]$\mathcal{N}or$ is a function used to perform a normalisation task.

[5]A conditional element is an element used in predicate to filter information.

[6]Content-join is a join represented by 'OR' or 'AND' between the element with different values or different element with different values.

[7]**XMLSpy** is an XML editor and integrated development environment (IDE) from www.Altova.com