Using Data in the Transformation of XML Queries

Ke Geng
Computer Science
University of Auckland
Auckland, New Zealand
ke@cs.auckland.ac.nz

Gillian Dobbie
Computer Science
University of Auckland
Auckland, New Zealand
gill@cs.auckland.ac.nz

ABSTRACT
XML query transformation is a key part of XML query optimization. However most research into transforming XML queries is schema-based rather than data-based because of the highly flexible nature of XML documents. In this paper, we introduce an engine that we designed and built to test the effectiveness of different data-based query transformation techniques. The information is extracted from the data or XML documents, and with this engine, queries can be automatically transformed if they fall into particular categories which we have defined. Because this engine is developed independent of any particular XML database and different functions are realized by different modules, the engine is extensible and easy to adapt for different database systems.

Categories and Subject Descriptors
E.0.e [Data]: Knowledge and data engineering tools and techniques; E.5.b [Data]: optimization

General Terms
Performance and Experimentation

Keywords
XML, query optimization

1. INTRODUCTION
XML query transformation, which is an important part of XML query optimization, has been paid more attention recently. Currently, most XML query transformation methods that have been developed are based on structural information that can be derived from a schema [16] [11] [10]. Research into XML query transformation based on information that can be derived from the XML document is limited by the flexibility and complexity of XML documents.

In this paper, we describe a transforming engine that we have built, to aid our research into transforming XML queries based on information derived from the XML document. In our method, XML document elements will be classified based on their value characteristics. Queries can be evaluated and transformed using the element classification information. To the best of our knowledge, no research into transforming XML queries based on element classification has been carried out to date. Also no transforming engine, which uses information about element classification in query transformation, has been designed. An engine is needed to examine the practicality of our method of query transformation and to test the influence on query execution. The engine we designed has the following functionality:

1. Extracting element information.
3. Choosing suitable descriptions and deriving classification information from the descriptions.
4. Evaluating input queries with classification information and choosing suitable transformations.

Because the transforming engine is designed as middleware that works between an application and a database system and is independent of any XML databases, it can be used in experiments on different XML databases, including both native XML databases and XML-enabled databases, which makes it a powerful and necessary tool for research.

The remainder of this paper is structured as follows. Some background information is introduced in section 2. The engine is described in detail in section 3. The experiments and the results are discussed in section 4. In section 5, the related work is discussed. Section 6 draws the conclusions and describes future work.

2. BACKGROUND
In this section, we discuss the existing semantic query optimization methods for XML queries and introduce the possibility of transforming XML queries using information derived from the data in XML documents.

XML semantic query optimization can be classified into two main groups: optimization based on information derived
from the document’s structure or schema and optimization based on information derived from the document’s data. Most XML semantic query optimization technologies that have been developed belong to the group of optimizations based on information derived from the structure and these technologies improve XML query execution in different ways. In [15], a method is introduced to classify elements with the information derived from the DTD [17]. The authors of [3] introduce a method for transforming XML queries using XML Schema. The method discussed in [5] checks target XML documents against the Schema [4] and skips unnecessary computations. The authors in [11] introduce a method to transform the XPath [6] queries based on the unique location path constraints extracted from the XML Schema.

Currently most research into optimizing XML queries based on information derived from the data concentrates on statistical information [12] [1]. The reason given for this is that XML documents are too flexible and therefore difficult to analyze. We demonstrate the flexibility in the example shown in Fig. 1. We can find “ID” elements embedded in both “vehicle” elements and “engine” elements. And the “ID” elements are distributed in different levels of the document. Also, the structure of each “vehicle” element is not same.

However if we concentrate only on specific elements instead of the whole document, some element value characteristics may be extracted. For example, if we only analyze elements “style” and “price” that are embedded in element “car”, we can find that all the prices of all luxury cars are higher than $100,000 and the prices of all classic cars are lower than $37,000. With this characteristic, we can classify elements “car” into two groups:

- **luxury car**: which are more expensive than $100,000;
- **classic car**: which are cheaper than $37,000.

We now give an example demonstrating how the information which is derived from the XML document can be used for query optimization. Consider the query

```
document('store.xml')//car[style = 'luxury' and price > 110000]
```

which returns all luxury cars which are more expensive than $110,000. This query can be simplified to

```
document('store.xml')//car[price > 110000]
```

since only luxury cars cost more than "110000".

3. TRANSFORMING ENGINE

In this section, we introduce the possible transformations based on information derived from the data. Then we describe the added functionality implemented in our transforming engine.

3.1 Transformation categories

In our method, all elements will be analyzed. And possible classification for elements will be carried out based on their subelements. Information of the classification will be recorded. The information that is retained with each classification includes: related element names, related elements’ value characteristics and the number of related elements in each group, since all these characteristics may be important for query transformation.

We discuss three transformations based on the elements’ classification that we have identified initially. In order to explain them clearly, we will use “Condition_M(e)” to represent the result of evaluating element “e” with query condition “M” and “σ_E[Condition_M(E)]” to represent all the elements that satisfy query condition “M” against the elements in E. The three transformations that we have identified to date based on information derived from the data are:

- **Elimination**
  Elimination is the operation that eliminates queries with mutually exclusive query conditions. This transformation will help systems to detect and block queries that do not return any results. Consider a query
  
  ```
document('store.xml')//car[style = 'luxury'
  and price > 110000]
  ```

  We will use “Condition_M(e)” to represent the result of evaluating element “e” with query condition “M” and “σ_E[Condition_M(E)]” to represent all the elements that satisfy query condition “M” against the elements in E. The three transformations that we have identified to date based on information derived from the data are:

- **Elimination**
  Elimination is the operation that eliminates queries with mutually exclusive query conditions. This transformation will help systems to detect and block queries that do not return any results. Consider a query

  ```
  document('store.xml')//car[style = 'luxury'
  and price > 110000]
  ```

  We will use “Condition_M(e)” to represent the result of evaluating element “e” with query condition “M” and “σ_E[Condition_M(E)]” to represent all the elements that satisfy query condition “M” against the elements in E. The three transformations that we have identified to date based on information derived from the data are:
Executing this query against the document shown in Fig. 1 will not return any results. If we know before execution that all luxury cars are more expensive than $100,000, we can return the answer $\emptyset$ and block the query before it is sent to the server. The reason this transformation works is based on this formula:

$$\text{Condition}_{M \land N}(e) = \text{False}$$

if $\text{Condition}_M(e) \land \text{Condition}_N(e) = \text{False}$

- **Reduction**

  Reduction is the operation that eliminates redundant query conditions. The transformation may improve query execution by reducing the number of constraints that are evaluated. Consider the query

  $$\text{document("store.xml")//car[style = 'classic' and price < 40000]}$$

  which can be transformed to

  $$\text{document("store.xml")//car[style = 'classic']}$$

  if it is executed against the document in Fig. 1 because only all classic cars cost less than 40000. The reason this transformation works is based on this formula:

  $$\text{Condition}_{M \land N}(e) \Rightarrow \text{Condition}_M(e)$$

  if $\forall e (e \in \sigma_{\text{Con}(M)}(E) \Rightarrow e \in \sigma_{\text{Con}(N)}(E))$

  and

  $$\text{Condition}_{M \lor N}(e) \Rightarrow \text{Condition}_N(e)$$

  if $\forall e (e \in \sigma_{\text{Con}(M)}(E) \Rightarrow e \in \sigma_{\text{Con}(N)}(E))$.

- **Introduction**

  Introduction is the transformation that adds a new condition, with the aim of reducing the query searching area, compared with the original query condition. Consider the query

  $$\text{document("store.xml")//car[price < 35000]}$$

  This query can be transformed to

  $$\text{document("store.xml")//car[style = 'classic' and price < 35000]}$$

  This transformed query could improve the speed of query execution by reducing the searching area. This transformation is more complex than the other two transformations and there are some necessary conditions for this kind of transformation:

  - the range of the original query condition must be a subset of the range of the added condition.

  

  For more discussion about each transformation please refer to [9].

---

**Figure 3:** The descriptions in OWL

```xml
<owl:Class rdf:ID="person_null_plug_age_30">
  <owl:Restriction>
    <owl:OnProperty rdf:resource="has_age"/>
    <owl:maxValue rdf:datatype="&xsd:string">29</owl:maxValue>
    <owl:minValue rdf:datatype="&xsd:string">18</owl:minValue>
  </owl:Restriction>
</owl:Class>
```

**Figure 4:** Schema for element “age”

- an index exists on the added condition-related element.
- the selectivity of the added condition-related element is higher than that of the original condition-related element.

The reason this transformation works is based on this formula:

$$\text{Condition}_M(e) \Rightarrow \text{Condition}_{M \land N}(e)$$

if $\forall e (e \in \sigma_{\text{Con}(M)}(E) \Rightarrow e \in \sigma_{\text{Con}(N)}(E))$

### 3.2 Transformation engine

The transformation engine that we designed and built includes the following modules: information management, information extraction, blocking unsatisfied query, reducing condition and introducing condition. The information management module is designed to manage the descriptions used in query transformation. The information extraction module is designed to extract information for each kind of element from the XML document. The last three modules are each in charge of each transformation discussed in section 3.1. The structure of the transforming engine is shown in Fig. 2.
3.2.1 Information management module
This module is designed to manage the descriptions for the transformations. In our method, all the classification information is stored in an OWL document and the OWL document is stored together with the XML documents. We choose OWL to store the classification information because OWL documents can be read and edited manually as well as automatically, and OWL provides different options for meaningful descriptions. A piece of description for information derived from data is shown in Fig. 3. From this description, we can extract the following information:

- the description is about the classification of element “person”.  
- the classification is based on two elements “age” and “totalPay”. Both of the elements are descendent elements of element “person”.  
- there are 5000 person elements in this classification.  
- the values of “age” elements are between 18 and 29;  
- the value of “totalPay” elements are between 5000 and 10000.

It would also be possible to store more detailed classifications, such as the totalPay of classes of age, where people below the age of 30 have pay less that 20000, people with age between 30 and 40 have pay between 20000 and 50000, and people over the age 40 have pay greater than 50000. The information in the OWL document is important for transformations. Deriving the information from the database for each query is unacceptable because the repeated read operation will occupy a great deal of precious resource of the database system. If possible, the information in the OWL document should be loaded into main memory when the transformation engine starts and should not be removed until the transformation engine stops. In this way, the repeated read operation can be avoided and the time spent on query transformation will be reduced. In our design, we use a multi-dimensional array to store all the information from the OWL document. Whenever the transformation engine starts, all the information in the OWL document will be loaded into the array.

3.2.2 Information extraction module
This module is designed to extract element information when an XML document is loaded into the database. The information extracted by this module include elements’ name, elements’ domain, elements’ domain styles and names of the descendent elements. The elements’ domain style records whether the domain is restricted or not. In our method, we assume the elements, whose domain is restricted, have higher selectivity than the elements whose domain is unrestricted. All the extracted information is stored together with the information about element classifications.

Usually, the element information extracted by this module can be found in XML Schema or DTD. However, we choose to extract the information because:

- Not all XML documents are equipped with XML Schema or DTD
- Sometimes the XML Schema or DTD do not reflect the exact information.

Consider the piece of XML Schema in Fig. 4, which defines element “age” embedded in element “person”. From this description, it can be derived that the domain of age is from “0” to “120”. But the real domain of element “age” may be smaller than the domain defined in the Schema. Suppose the real domain of “age” is from “20” to “65”, then the query

\[
\text{document('record.xml')//person[age > 70]}
\]

would return no results. We could not conclude that the query would return no results based on the schema, but if we extract the real domain from the XML document, we can evaluate this query and block it before it is sent to the server.

3.2.3 Blocking unsatisfied query module
An unsatisfied query is a query that follows the grammar of the query language but does not return any result because of the query conditions. Module “blocking unsatisfied query” is designed to detect four kinds of unsatisfied queries:

- Queries with conditions about elements that do not exist in the document;  
Consider a query

\[
\text{document('record.xml')//PDA}
\]

If there is no “PDA” element embedded in XML document record.xml, then the query would not return any result.

- Queries with conditions about paths that do not exist in the document;  
Consider an XML document where there is no relationship, either direct and indirect, between two elements “person” and “car”.  
The query

\[
\text{document('record.xml')//person[car]}
\]

is looking for “person” elements that have subelements “car”, so no result is returned.
Queries whose query condition is outside an element’s domain;
The query
\[
\text{document(‘record.xml’)//person[age > 200]}
\]
belongs to this class of queries.

- Queries with mutually exclusive query conditions.

The treatments for the first three kinds of queries have been discussed in [7]. In this paper, we discuss the procedure of detecting the query with mutually exclusive query conditions. The procedure can be represented in the following steps:

1. The engine selects query related descriptions that include the elements that appear in the query from the array. If no description is found, the query will be sent to the database.
2. The engine chooses the description, whose domain intersects with the domain of one of the query conditions. Consider the following three classification descriptions:
   - description 1: all persons in this group are younger than 30 and have a deposit less than $5000.
   - description 2: all persons in this group have age between 30 and 40, and have a deposit between $5000-$10000.
   - description 3: all persons in this group have age between 40 and 50, and have a deposit between $10000-$15000.

Consider also the input query
\[
\text{document(‘record.xml’)//person[age > 37 and deposit < 4500]}
\]

It can be seen that the elements in the three descriptions coincide with the elements in the conditions in the input query. The engine checks the query conditions against each of the descriptions in turn. Because none of the descriptions subsume the query conditions, the query is identified as an unsatisfied query.

When an unsatisfied query is detected, this module will inform the users and discard the unsatisfied query. This module is always executed before the other two modules because executing this module first will save system time by avoiding further operations on unsatisfied queries.

3.2.4 Reducing condition module
Module “reducing conditions” is designed to implement the “reduction” transformation in query optimization, removing redundant query conditions. This module transforms queries in three steps and the first two steps are the same as the first two steps in “blocking unsatisfied query”. Here we only introduce the third step in detail.

1. The engine selects query related descriptions for the elements whose name appears in the query from the array. If no description is found, the query will be sent to the database.
2. The engine chooses the description, whose domain intersects with the domain of one query condition.
3. The transforming engine checks the domain of the query conditions and reduces the query condition which has a smaller domain if the operator between the conditions is “or”. If the operator between the two conditions is “and”, the condition with the bigger domain will be removed. If the domain of the two conditions is the same, the transforming engine will choose the query condition, which has a higher selectivity.

We use an example to explain the third step. Consider a description that specifies that all the luxury cars are more expensive than $100,000 and a query
\[
\text{document(‘record.xml’)//car[style = ‘luxury’ and price > 150000]}
\]

With the description, the engine will remove a condition and transform the query to
\[
\text{document(‘record.xml’)//car[price > 150000]}
\]

The engine will remove the condition and transform the query to
\[
\text{document(‘record.xml’)//car[price > 150000]}
\]
because element “style” has a higher selectivity than element “price”.

3.2.5 Introducing condition module

The module “introducing condition” is designed to carry out the “introduction” transformation. If multiple descriptions can be found that would reduce the workspace, the engine will calculate the domain of each possible transformation and introduce the condition which has the smallest domain. This module transforms queries using the following steps:

- The engine selects all related descriptions and stores them in an array. The description that is chosen must belong to an element that has high selectivity. Whether an element has high selectivity is evaluated with the elements’ value style, which is extracted by the “information extraction module”. For example, for the query

  \[
  \text{document} \left( \text{record.xml} \right) \left/ \text{car} \left[ \text{style = 'luxury'} \right] \right.
  \]

the engine may choose descriptions related “car”, “price” and “style”. Descriptions related with “car”, “price” and “year” will not be considered because different cars may have many different values of “year”.

- The engine chooses the description, whose domain intersects with the domain of one query condition.

- The engine chooses a description, which has the smallest domain, and introduces the description as an additional query condition.

4. EXPERIMENTATION

In this section, we present the experiments designed to test the performance of the transforming engine. All the experiments are run on an HP workstation XW4200, which is equipped with an Intel Pentium 4 CPU 3.40GHZ and 2 GB memory.

The experimental data sets are built using an XML generator [8] based on the XMark benchmark [14]. We use the generator to build five XML documents with sizes of 20M, 40M, 60M, 80M and 100M. Then we insert four kinds of elements as child nodes to the “person” element. The inserted elements and values are listed in Fig. 5. From the figure, you can see that “person” elements can be classified into different groups based on the values of the inserted elements. For instance the youngest person with ages between 18 and 29 spend $100 to $5000. Their importance value is 1 and they are arranged in area 1.

Three groups of queries are designed for the experiment and the formats are shown in Fig. 6. For module “blocking unsatisfied query”, four queries are designed to test the performance on detecting four kinds of unsatisfied queries. Because the four queries will be transformed to $\emptyset$, there is no query generated. For “simplification” and “multiplication”, we can get a series of queries with different selectivity by changing the value of $V_{al_1}$ and $V_{al_2}$. The introduced condition for “multiplication” will be generated automatically according the descriptions of element classifications and for different situation the generated condition are different.

We choose eXist[2] as the database in our experiments. Because eXist provides a query caching function to improve the query execution and indexes on elements can be built either automatically and manually, both the original queries
and the generated queries are executed in four situations:

a. no query caching plus an automatically built index,
b. no query caching plus a manually built index,
c. with query caching plus an automatically built index and 
d. with query caching plus a manually built index. Because of space limitations, we only list the results from two situation: no query caching with an automatically generated index and with query caching and a manually generated index, and with two selectivity: 20% and 80%.

**Experiment for module “blocking unsatisfied query”**

The results of these experiments are shown in Fig. 7. It can be seen that the resources of the database can be saved greatly by blocking the unsatisfied queries. This is because all the computations are completed on the client side, and there is no time spent on transferring data nor query execution on the server side. In contrast to the time spent on the execution of the original query, the time spent on detecting unsatisfied query is very short and can almost be ignored.

**Experiment for module “reducing condition”**

The performance of “reducing condition” is shown in Fig. 8. It can be seen that the performance of query execution can be improved by about 30% by the module “reduce condition” when the query is executed with the automatically built index. When the queries are executed with a manually built index, the performance can be improved by around 20%. This phenomena becomes more obvious when the selectivity or the size of the XML document is increased.

**Experiment for module “introducing condition”**

From Fig. 9, it can be seen that the queries, which are generated by module “introducing condition”, execute slower than the original queries. This is different from our prediction. In fact, the bigger the selectivity or the size of the document, the more the execution time increase. By analyzing the results of the experiments, we assume the module “introducing condition” may only work on query engines which execute sub-query execution strategy. And the module “introducing condition” may not work with query engines, which execute queries by choosing the intersection of the results of sub-queries.

From the experiments we can see that the engine is a powerful and necessary tool for the research of XML query transformation, which can help us to evaluate the performance of different transformation methods.

## 5. RELATED WORK

In this section, we present several technologies for XML semantic query optimization.

In [15], an element classification method is introduced, which is based on structural information. By analyzing the DTD [17] of an XML document, elements will be classified based on their child nodes. For example, the piece of a DTD "< !ELEMENT person( name, email)+ >" will lead to a classification of “person with email” and “person without email”. With this classification, the query searching area will be reduced and the query execution performance is improved.

Authors of [3] introduce a method of heuristic-based algebraic XML query transformations. The method is based on a series of equivalences that are represented in PAT algebra expressions [13]. With information about the structure of an XML document, a set of deterministic algebraic transformation rules can be derived based on PAT equivalences. Then input queries can be transformed to an effective one with the transformation rules.

In [10], an ordered Schema graph is introduced, which is based on the XML Schema of the destination XML document. Then XQuery graphs are generated from the ordered Schema graph for each input query using their XPath expressions. By marking all nodes within the XQuery graph that contribute to a successful evaluation of each sub-expression, sub-expressions of queries, which do not generate any output or generate superfluous intermediate results, will be identified and eliminated.

All the work described in this section improves the speed of XML query execution. However, none of them use information about the data such as value distribution in the query transformation.

## 6. CONCLUSION AND FUTURE WORK

In this paper, we introduce a transforming engine designed for our research into query transformation based on element classification. With this engine, all input XML queries will be evaluated with the information extracted from the
data and possible transformations are carried out automatically. Another important function of the query transformation engine is information management. Whenever the engine starts, all information is loaded into main memory and for each query query-related descriptions are analyzed and chosen automatically.

Currently the element classification is carried out manually and the related descriptions are edited manually. An analyzing module will be built to analyze the value characteristics of elements and classify elements based on the value characteristics. A description generator will also be built to generate descriptions for elements and all descriptions will be stored in an OWL document. More functionality, such as description maintenance and description updating, will be added to the module “information management”. Lastly, we will carry out experiments to measure the effectiveness of the transformations for query optimization.

7. REFERENCES