Multi-XPath Query Processing in Client-Server Environment

Ren Yan

Abstract

When a client submits a set of XPath queries to an XML database across a network, the answers sent back by the server may include redundancy because of the characteristics of XML and XPath: XML data has a nested structure and XPath query retrieves substructures appearing at arbitrary levels. This kind of redundancy arises in two ways: some elements may appear in more than one answer sets, or some elements may be subelements of other elements. In this thesis, we propose an algorithm to eliminate this kind of redundancy in multi-XPath query processing by replacing redundant data with pointers. In particular, two different approaches are designed for pointer insertion. It is shown in experiments that this approach can substantially reduce the communication costs in multi-XPath query processing in a client-server environment, which is critical in slow networks where the communication cost could easily become a bottleneck.

Acknowledgement

I would like to thank my supervisor, Dr. Chan Chee Yong for his guidance and encouragements through the whole project.

Contents

Summary

When a client submits a set of XPath queries to an XML database across a network, the answers sent back by the server may include redundancy because of the characteristics of XML and XPath: XML data has a nested structure and XPath query retrieves substructures appearing at arbitrary levels. This kind of redundancy arises in two ways: some elements may appear in more than one answer sets, or some elements may be subelements of other elements. In this thesis, we propose an algorithm to eliminate this kind of redundancy in multi-XPath query processing by replacing redundant data with pointers. In particular, two different approaches are designed for pointer insertion. It is shown in experiments that both two approaches can substantially reduce the communication costs in multi-XPath query processing in a client-server environment, which is critical in slow networks where the communication cost could easily become a bottleneck.

1 Introduction

1.1 Motivation

As XML has gradually become the standard for information representation and interchange on the Internet, there have been many researches of XML information exchange on networks. In general, those services can be classified into two categories: those that process queries on the server side, such as online XML databases and continuous query systems, and those that process queries on the client side, such as XML streaming systems.

Most XML information services use some kind of query language and among them XPath has become the most popular. XPath is originally designed to be used by XSLT and XPointer, but it is now also used as an independent query language for many XML information systems. XPath is a tree pattern language which selects nodes from XML data based on their structure. Unlike some full-fledged query language like XQuery, it only extracts a whole subtree rooted by some node without any modification. This property is the reason why XPath is more efficiently processable and hence has become probably the most successful XML technology besides XML itself. However, it is also this characteristic of XPath that causes the data redundancy problem which we are going to solve in this thesis.

In a client-server system, when a client submits a set of input queries to server, the answer sets sent back by the server may include redundancy caused by the nested structure in XML data. In some case, the answer sets may be even larger than the database itself. This kind of redundancy occurs in two ways:

1. Some elements may be included in more than one answer sets

   For example, when a client submits two queries to a bookstore database asking for: 1) all books in English 2) all books in English or French, elements representing English books will appear in answer sets to both queries.

2. Some elements may be subelement of other elements

   For example, when a client submits two queries to a bookstore database asking for: 1) all shelves 2)all books on shelf No. 21, every element in the answer set for query 2 is a subelement of some element in the answer set for query 1.

   Moreover, even when a client submits a single query, the answer returned by the server may be self-redundant when it addresses a part of XML data with recursive structure. For example, suppose the client submits a query "//a" to the server, it will retrieve all the subtrees

rooted by "a" nodes. Therefore, if some "a" occurs as descendants of other "a", the subtree rooted by descendant "a" is sent more than once over the network.

As a result, answer sets to this kind of queries could be very large due to redundancy. In this case the communication cost could become a bottleneck as the network speed is usually slow in a server-client paradigm.

A lot of research work has been done in recent years to reduce communication costs in the context of XML databases. In particular, K. Tajima et al. proposed a minimal view approach in [27] to solve the redundancy problem caused by nested structure of XML.

1.2 Tajima's Client-based Approach

K. Tajima et al. [27] proposed an algorithm to eliminate redundancies by sending minimal views. Figure 1 illustrates how their approach works: given a set of input XPath queries $\{Q_1, ..., Q_n\}$, the pre-processor at the client side computes a view set $\{V_1, ..., V_m\}$ which will retrieve all the necessary information asked by $\{Q_1, ..., Q_n\}$, and a triplet list which indicates how to derive the real answers out of the answers to the views. After the server

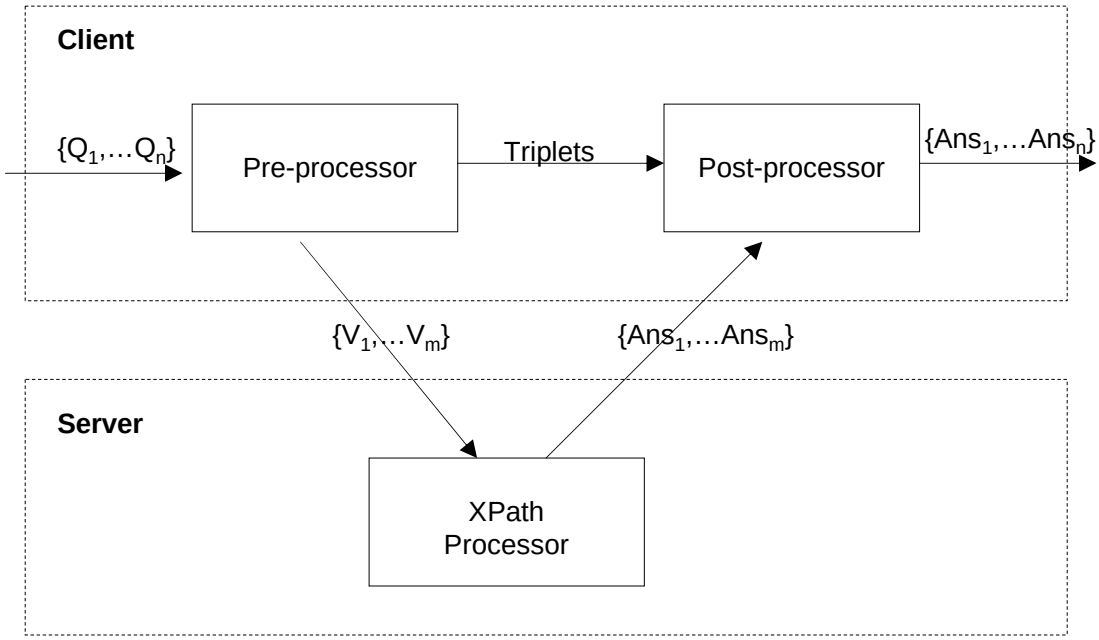

Figure 1: System diagram of Tajima's client-based approach

receives this view set, it simply evaluates them and sends the answer set $\{Ans_1,...,Ans_m\}$ back to the client. The client then compute the real answer set out of $\{Ans_1,...,Ans_m\}$ and the triplet list.

The answer set $\{Ans_1,...,Ans_m\}$ to the views is guaranteed to be minimal as it only contains elements that appear in the final answer $\{Ans_1,...,Ans_n\}$ and each element appears only once.

As the descendant axis "//" represents a restricted form of recursion, queries with "//" is called recursive queries while queries without "//" is called non-recursive queries. In the pre-processor phase, different methods

are proposed for different types of XPath queries. An automata-based algorithm is designed for non-recursive queries since they can always be translated into acyclic deterministic finite automata. On the other hand, the deterministic finite automata derived from recursive queries are inevitably cyclic and therefore a method based on set operations was proposed for recursive queries instead. More details are given in Chapter 3.

1.3 Contributions

In this thesis, we have the following contributions:

- We propose a server-based approach to optimize multi-XPath query processing in a client-server environment with respect to the communication cost.
- We propose two different methods to replace the redundant data with pointers: the *Embedded Pointer* approach and the *Separate Pointer* approach.
- We implement both *Embedded Pointer* approach and *Separate Pointer* approach.
- We implement Tajima's approach [27] as the main reference work.

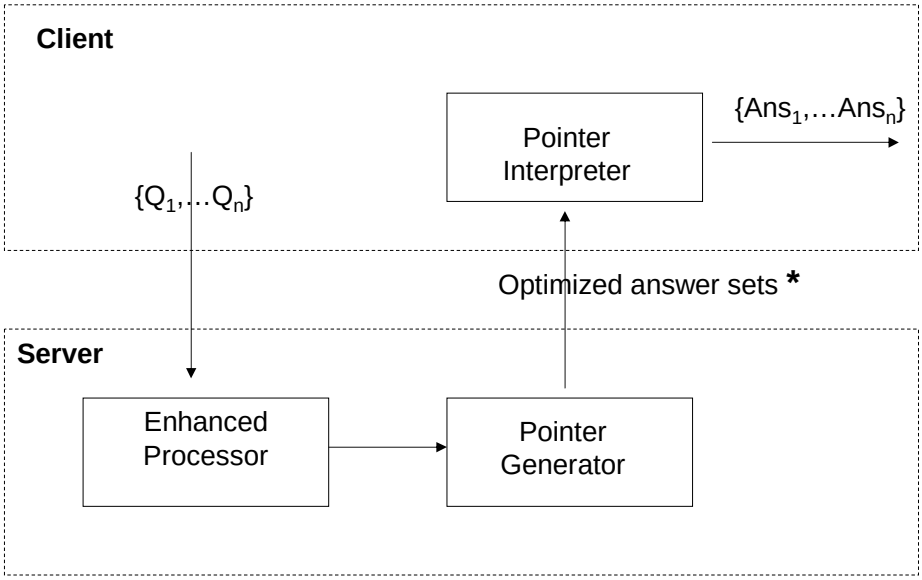

Figure 2: System diagram of our server-based approach

- We conduct various experiments to test and compare the performance of both methods and Tajima's approach.

Tajima's algorithm can be considered as a client-based approach as their main effort to eliminate redundancy is made by the pre-processor and post-processor at the client side, whereas the server side only has a dummy XPath processor. On the other hand, we propose a server-based approach to solve the same problem. Our approach removes the redundancy during query processing at the server side by making answer sets to different queries share their intersections with the help of pointers.

The main procedures of our approach is shown in Figure 2: when the server receives a set of input XPath queries $\{Q_1, \ldots Q_n\}$ submitted by the client, an enhanced XPath processor evaluates them and gets a set of distinct answer nodes. A pointer generator then outputs a set of optimized answer sets with redundant data replaced by pointers. Once the client receives the optimized answer sets, it invokes a pointer interpreter to retrieve the original data represented by the pointers. Basically a pointer is a tag which indicates how to retrieve the original data. Two different methods are designed for the pointer generator. As their names suggest, the *embedded pointer* method produces a set of answer files with pointers embedded in; the *separate pointer* method produces a text file and a set of pointer files. More details are given in Chapter 4.

We have implemented both methods of server-based approach and Tajima's client-based approach. The experimental results have been compared and reported. It shows our server-based approach could indeed minimize the communication costs, which is critical in low/medium speed or high traffic network.

1.4 Outline

The rest of this report is organized as follows. In the next chapter we give a short review of related work, whereas Tajima's client-based approach, as our main reference, is surveyed in more detail in Chapter 3. In Chapter 4 we presented our own server-based approach. The experimental results are reported and compared in Chapter 5. Finally, our work is summarized in Chapter 6.

2 Related Work

In this chapter we give a review of three research areas that are related to this thesis. They are single query optimization, multi-query optimization and minimizing communication cost in client-server model.

2.1 Single Query Optimization

As both XML and XPath becomes more and more popular nowadays, a variety of techniques have been developed to speed up XPath query evaluation, such as indexing and query rewriting.

The typical methodology of XML indexing is to first construct a graph-based equivalent of the original XML document, and then to create indexes on this graph representation. *Lore* system [20] is a cost based query optimizer, which represents early work on storing and querying semi structured and XML data. *Lore* uses a combination of techniques for query processing, particularly relying on a DataGuide [13] as a structural summary used to discover path and tree patterns. DataGuides are a concise and accurate summary of all paths in the database that start from the root. It describes every unique label path of a source exactly once, reducing the portion of the database to be scanned for path queries. *Lore* contains several indexing

structures that are useful for navigating the database. They are value index, label index, edge index and path index. *Lorel* queries can be compiled into query plans that make efficient use of the indexes.

More novel indexing schemes are proposed recently. The Index Fabric [5] employs a string index to solve containment queries. APEX [4] is an adaptive path index, using data mining algorithms to summarize paths that appear frequently in the query workload. XISS [16] adopts a numbering scheme for elements in the hierarchy of XML data, which can be used to quickly determine the ancestor descendant relationship and expedite the query expressed in regular path expressions. ViST [30] transforms both data and queries into structure-encoded sequences to avoid expensive join operations.

The technique of rewriting queries using views to speed up query evaluation has been well studied in the context of relation database [14]. Recently this technique is used to optimize regular path queries in semi-structured database [2, 12]. Most recently [32] proposes an algorithm to find minimal rewritings, which is reported to be complete and sound for a fragment of XPath. The technique of query rewriting using materialized views is also widely studied in client-server model and will be discussed in Chapter 2.3.

2.2 Multiple Query Optimization

As database systems often need to execute a set of related queries which may share common subexpressions, the multi-query optimization (MQO) problem becomes an important concern in many application domains, such as relational databases, deductive databases, decision support systems, and data analysis applications. Basically multi-query optimization is a technique that allows a set of queries to be computed together by detecting their similarities. Its objective is to exploit the common subexpressions between a set of queries to be executed concurrently and reduce the execution time by reusing the cached results that have been previously computed.

After Sellis presented the first systematic analysis of MQO problem in [23], this problem has been well studied in the context of relational database over the past seventeen years. The researches before [23] were simply based on the idea of reusing temporary results from the query execution, while the processing of each individual query is based on a locally optimal plan. However, the union of locally optimal plans does not necessarily form a globally optimal plan, hence [23] proposes a heuristic algorithm to exhaustively find a global optimal query plan between a small number of queries. An extended improved algorithm was then proposed in [24] to search for the global opti-

mal plan in the state space that models all alternatives for evaluating a batch of queries. Both [23] and [24] only examines a fraction of all possible global processing plans and may lose some potentially good plans.

Recent works provide heuristics for reducing the search space. [22] proposes three cost-based heuristic algorithms, among which the greedy heuristic adopts various optimizing techniques that improves efficiency significantly. [17] proposes an optimization for multiple view maintenances by using intermediate views with common subexpressions. As traditional techniques rely on materialization of the common subexpressions to avoid recomputing shared results, [9] pipelines the common subexpressions to avoid unnecessary data materialization. The authors show that finding an optimal materialization strategy is NP-hard and present a greedy heuristic for finding good strategies. [10] proposes a new approach for multi-query optimization that uses middleware to queue and schedule the input queries to form synchronous groups and teams.

2.3 Minimizing Communication Cost In Client-Server Environment

Network performance is always an important concern for a client-server system where large communication costs can easily become a bottleneck. Therefore the problem of optimizing communication costs in query processing over a network has grabbed significant attention. Various techniques have been proposed to reduce communication costs in the context of different databases, such as view selection and data caching.

The traditional view selection problem is to find efficient methods of answering a query using a set of pre-defined materialized views over the database. There have been many researches in this problem because it is relevant to a lot of data management problems such as query optimization, data integration and data warehouse design. Recently a similar idea has been used to optimizing communication costs in the context of relational databases. [3] discusses the general problem of finding optimal view sets to answer a workload of conjunctive queries. In this paper, the authors shows that disjunctive view sets are considered to be an optimal solution when the query has self-joins. For the queries without self-joins, they also proposed a dynamic-programming algorithm for finding optimal disjunctive view sets.

In [15] the same authors present more techniques for reducing the size of the search space of views and for efficiently and accurately estimating the sizes of views.

Data caching at local client plays an important role in improving the performance of client-server systems. The basic intuition of data caching is to effectively utilize the storage resource in the local client to cache the results of the prior queries for possible later reuse. The concept of semantic caching was proposed in [8] and [21]. In semantic caching, the client caches a semantic description of the data instead of a list of physical tuples or pages which are used in conventional caching. When a user issues a new query, the client makes use of the semantic descriptions to determine what data are locally available in its cache, and submits a remainder query to retrieve data which are not overlapped with answers of any prior queries.

Technique of caching popular queries and reusing results of these previously computed queries is firstly studied in the context of relational database. It is crucial for good performance of distributed environments such as the Web. [33] developed a customizable cache system that caches data at different levels according to the web site's different content. This system reduces the costly interaction with databases and therefore improves the response

times from data-intensive web sites.

Lately semantic caching has attracted a lot of attentions in the context of XML database. [7] proposes a semantic cache of XQuery views based on query containment and rewriting techniques. [19] proposes a novel view-based caching strategy. It maintains a semantic cache of materialized XPath views, which are stored in relational tables and are accessed by SQL queries. Therefore the cache lookup is very efficient. It also adopts the technique of XPath query containment [18] to decide if a given query can be answered by a cached view. Both [19] and [7] can only answer queries whose results have already been cached. On the other hand, [31] proposes a semantic caching system that can use its cached data to answer new queries that may not be cached. It caches XML data in tree structure with a semantic scheme, which consists of a set of patterns. When an XML query is received, the local client decides whether the cached XML tree is able to totally answer this query according to current semantic scheme.

Most existing techniques, including [7, 19, 31], focus on how to reuse the answers received for previously processed queries. They only consider the redundancy between different transmissions but neglect the possible duplications within one transmission of query set. Moreover, these techniques do

not consider the redundancy caused by nested structure of the XML data, i.e., some answers appearing as substructure of other answers. However both this thesis and our main reference [27] concentrate on the duplications occurring within one single transmission between server and client, including the redundancy caused by nested structure of the XML data.

3 Client-based Approach

[27] is the most similar work to ours, therefore we take it as our main reference and have an entire chapter to survey it in detail. The authors of [27] make use of the technique of computing minimal views to reduce the communication costs in the context of XML database and XPath query. We call it client-based approach because their main effort to eliminate redundancy is made by the pre-processor and post-processor at the client side. In this chapter, we introduce the three methods proposed in [27] and give a brief discussion of their limitations.

3.1 Problem Formulation

The minimal view selection problem is formulated as follows: given a set of XPath queries $\{Q_1, Q_2, ..., Q_n\}$, it computes another set of XPath queries $V = \{V_1, V_2, ..., V_m\}$ such that:

1. V can answer all of $Q_1, Q_2, ..., Q_n$

2. among all possible candidates satisfying 1, the total size of the answers of $V_1, V_2, ..., V_m$ is minimal against any XML source.

In this paper, the input queries is restricted to a fragment of XPath language without ancestor axis, union and difference. The syntax of intermedia queries during processing is defined as follows:

$$q ::= /p|//p|q \bigcup q|q - q$$
$$p ::= a|\overline{\{a_1, ..., a_n\}}| * |p/p|p//p|p[p]|p\overline{[p]}$$

A query q is either an absolute location path of the form $/p$ or $//p$, the union of two queries $q \bigcup a$ or the difference of two queries $q - q$. a is a label test that matches nodes with a label a, an $\overline{\{a_1, ..., a_n\}}$ is a negative label test that matches nodes with a label other than $a_1, ..., a_n$.

The answer to an Xpath query is assumed to be given in the form of an XML tree rooted by a node labelled **Ans**. When a query answer is the following set of three subtrees: $\{< a > ... < /a >, < b > ... < /b >, < b > ... < /b >\}$, it is given as an XML tree in a form: $<$ **Ans** $> \{< a > ... < /a >, < b > ... < /b >, < b > ... < /b >\} < /$**Ans** $>$.

Given a set of queries $Q_1, ..., Q_m$, the algorithm needs to compute another set of queries $V_1, ..., V_m$ for the minimal view set, and a list of triplets showing how to extract answers to the original queries. Every triplet is of the form $Q \leftarrow (V, q)$, it means query q can be evaluated against the answer to query

V to retrieve part of answer to query Q.

The problem is solved step by step, the authors firstly gave an algorithm for non-recursive queries, followed by another algorithm for a single recursive query, and finally came out with an algorithm for the general case.

3.2 Non-Recursive Queries

For non-recursive queries, redundancy only appears between answer sets to different queries. An automata-based algorithm is designed for this type queries since they can always be translated into acyclic deterministic finite automata.

The algorithm first translates queries into deterministic automata, add fail states explicitly and construct a product automaton, which is a cross product of the automata. For each satisfiable path X from $(s_1, ..., s_n)$ to a state T of the form $(..., e_{i1}, ..., e_{i2}, ..., e_{ia}, ...)$ that does not go through any other states of the form $(..., e_j, ...)$:

1. add X to $V_1, ..., V_m$, and add $Q_i \leftarrow (X, /Ans/*)$to the triplet list for each $i \in i_1, ..., i_a$. Here X is the intersection of each query $\in \{Q_{i1}, Q_{i2}, ..., Q_{ia}\}$.

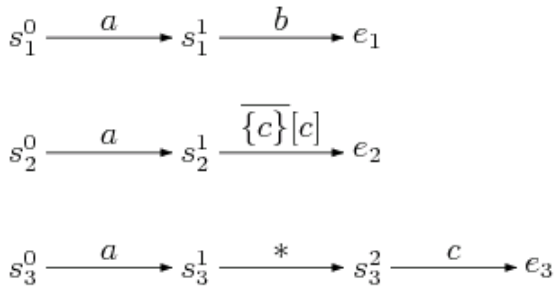

Figure 3: Automata for Q_1, Q_2, Q_3

This step ensures that every element only appear once in the entire answer sets.

2. for each path Y from the state T to any state of the form $(..., e_j, ...)$, if X/Y is satisfiable, add a triplet $Q_j \leftarrow (X, /Ans/*/Y)$ to the triplet list. Here path X/Y matches the subelements of the elements matched by X. This step ensures that no element is subelement of any other element in the entire answer sets.

For example, given three queries: $Q_1 : /a/b, Q_2 : /a/\overline{\{c\}}[c]$ and $Q_3 : /a/*/c$. They are first translated into three automata as shown in Figure 3. After adding a fail state to each automaton, a product of Q_1, Q_2 and Q_3 is constructed by computing the intersection and the difference between symbols. The product automaton is shown in Figure 4.

In this example, the algorithm produce a view set:

$V_1 : \{/a/b\overline{[c]}\}$

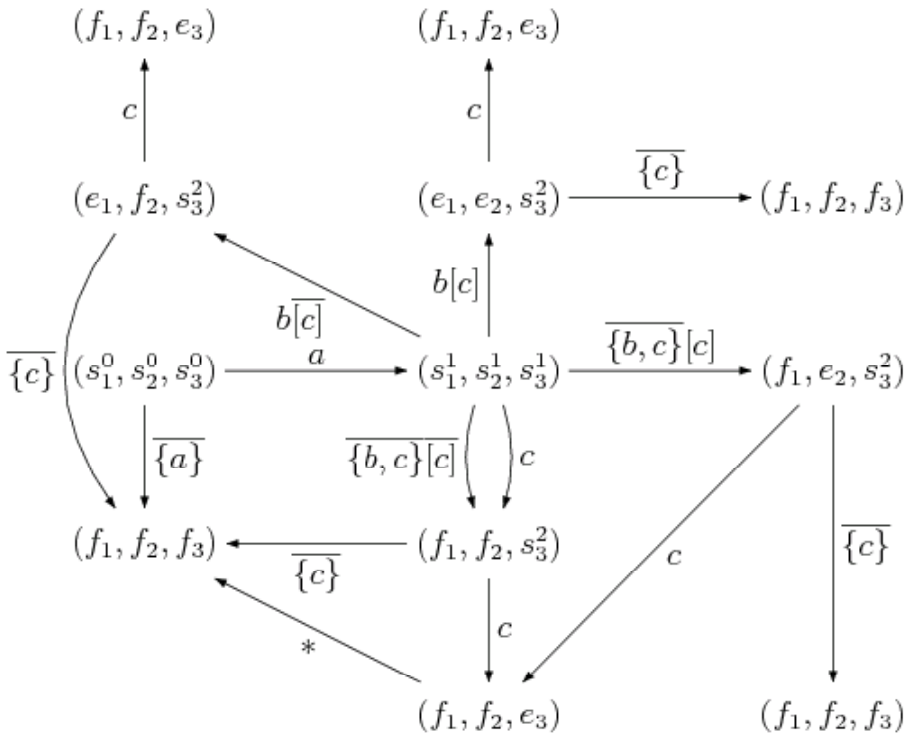

Figure 4: Product Automaton for Q_1, Q_2, Q_3

$V_2 : /a/b[c]$

$V_3 : /a/\overline{\{b, c\}}[c]$

$V_4 : /a/c/c$

and the following triplets:

$Q_1 \leftarrow (V_1, /Ans/*)$

$Q_1 \leftarrow (V_2, /Ans/*)$

$Q_2 \leftarrow (V_2, /Ans/*)$

$Q_2 \leftarrow (V_3, /Ans/*)$

$Q_3 \leftarrow (V_4, /Ans/*)$

$Q_3 \leftarrow (V_2, /Ans/ * /c)$

$Q_3 \leftarrow (V_3, /Ans/c)$

3.3 Single Recursive Query

The answer set to a recursive query might contains self-redundancy because of the nested structure of XML. In this case, the redundancy in the answers occurs even when only a single query is submitted. Unlike non-recursive queries, recursive queries can not be translated into a simple sequence of states, therefore the authors proposed a different algorithm for this kind of queries.

Consider a recursive query of the form: $Q : /p_1//p_2//.../p_n$, the redundancy in the answer occurs in two ways.

1. there are elements that match $/p_1//.../p_n//p_n$

   This kind of redundancy can be solved by simply submitting a view query: $(/p_1//.../p_n) - (/p_1//.../p_n//*)$ and applying $/Ans/*$ and $/Ans//pn$ to extract the final answer.

   For example, given a query $//a$, query $(//a - //a//*)$ is sent to the server to retrieve ***a*** nodes which occur as the first ***a*** node in each path.

2. there are elements that match $/p_1//.../p_n/p$ where p is some suffix of p_n such that the remaining prefix of p_n matches the suffix of p_n.

   For example, given a query /a//a/b/a/b, if there exist elements that match /a//a/b/a/$b_{(1)}$/a/$b_{(2)}$, the redundancy occurs as both $b_{(1)}$ and $b_{(2)}$ match /a//a/b/a/b.

   To remove this kind of redundancy, a set of relative location paths is considered: $S = \{*/p_n^{(1,k-1)}, */*/p_n^{(1,k-2)}, ..., */.../*/p_n^{(1,2)}\}$, where k is the length of p_n, and $p_n^{(i,j)}$ is the subsequence of p_n from position i to position j.

For every $T \subseteq S$, the algorithm computes the following views:

$$V(T) : (/p_1//.../(p_n \cap \bigcap_{p \in T} p - \bigcup_{p \in S-T} p)) - /p_1//.../p_n//*$$

If the result of $p_n \cap \bigcap_{p \in T} p - \bigcup_{p \in S-T} p$ is not empty, add $V(T)$ to the view set and the following triplets to the triplets list:

$(Q, V(T), /Ans/*)$

$(Q, V(T), /Ans//p_n)$

$(Q, V(T), /Ans/*/p_n^{(i+1,k)})$ for each $*/.../*/p_n^{(1,i)} \in T$

Intersection and difference of local paths with same length are computed with the help of product automaton:

Intersection $Q_i \cap Q_j$ is a union of queries corresponding to all satisfiable paths from $(s_1, ..., s_n)$ to any states of the form $(..., e_i, ..., e_j, ...)$. For example, the intersection of Q_1 and Q_2 is /a/b[c], path $(s_1^0, s_2^0, s_3^0) \longrightarrow (s_1^1, s_2^1, s_3^1) \longrightarrow (e_1, e_2, s_3^2)$.

Difference $Q_i - Q_j$ is a union of queries corresponding to all satisfiable paths from $(s_1, ..., s_n)$ to any states of the form $(..., e_i, ..., s_j^k, ...)$ where $s_j^k \neq e_j$. For example, $Q_1 - Q_2$ is $/a/b\overline{[c]}$, path $(s_1^0, s_2^0, s_3^0) \longrightarrow (s_1^1, s_2^1, s_3^1) \longrightarrow (e_1, f_2, s_3^2)$.

For the example /a//a/b/a/b, the set S includes */a/b/a, */*/a/b, */*/*a, the algorithm finally produces a non-empty view, resulting from $/a//(a/b/a/b \bigcap */* /a/b - */a/b/a - */*/*/a) - /a//a/b/a/b//*$:

V_1: /a//a/b/a/b/ - /a//a/b/a/b//*

and three triplets:

$(V_1, /Ans/*)$

$(V_1, /Ans//a/b/a/b)$

$(V_1, /Ans/*/a/b)$

3.4 General Case

In general case, the input query set may contain both recursive and non-recursive queries. Given the following set of queries:

$Q_1 : /_1^1 p_1^1 /_1^2 p_1^2 ... /_1^{l1} p_1^{l1}$

.

.

.

$Q_n : /_n^1 p_n^1 /_n^2 p_n^2 ... /_n^{l1} p_n^{l1}$

where p_i^j is an expression which includes neither / nor //, and each $/_i^j$ is either / or //. Prefix paths $pp_i^j (1 \leq i \leq n, 0 \leq j \leq l_i - 1)$ are defined as follows:

$$pp_i^j \equiv /_i^1 p_i^1 .../_i^j p_i^j \qquad \text{if } /_i^{j+1} = /$$

$$(/_i^1 p_i^1 .../_i^j p_i^j) \cup (/_i^1 p_i^1 .../_i^j p_i^j//*) \qquad \text{if } /_i^{j+1} = //$$

$$\emptyset \qquad \text{if } j = 0,\ /_i^1 = //$$

$$//* \qquad \text{if } j = 0,\ /_i^1 = //$$

For each S,T such that $S \subseteq 1, ..., n$, $S \neq \emptyset$, $T \subseteq (i,j)|1 \leq i \leq n, 0 \leq j \leq l_i - 1$, a view is computed as below:

$$V(S,T) : (\bigcap_{i \in S} Q_i - \bigcup_{i \notin S} Q_i) \cap (\bigcap_{(i,j) \in T} pp_i^j - \bigcup_{i \notin T} pp_i^j) - \bigcup_{1 \leq i \leq n} Q_i//*$$

Here $(\bigcap_{i \in S} Q_i - \bigcup_{i \notin S} Q_i)$ ensures that no element is shared by more than one answer set; $- \bigcup_{1 \leq i \leq n} Q_i//*$ ensures that only top-most answers are returned; $(\bigcap_{(i,j) \in T} pp_i^j - \bigcup_{i \notin T} pp_i^j)$ avoids redundancy of type 2 for recursive query.

For each $V(S,T)$, the following triplets are added to the triplets list:

$$Q_i \leftarrow (V(S,T), /Ans/*) \qquad \text{for } i \in S$$

$$Q_i \leftarrow (V(S,T), /Ans/* /_i^{j+1} p_i^{j+1} .../_i^{l_i} p_i^{l_i}) \qquad \text{for } (i,j) \in T$$

For example, given two queries $Q_1 : /a//b$ and $Q_2 : /a/b$, four prefix paths are generated by the algorithm: $pp_1^0 = pp_2^0 = \emptyset$, $pp_1^1 = /a \cup /a//*$ and $pp_2^1 = /a$. Since $\emptyset$ only creates empty set in set intersection and is

meaningless in set difference, only pp_1^1 and pp_2^1 are considered. Therefore 3 sets for S and 4 sets for T are used to produce 12 views:

$V_1 : (/a//b - /a/b) - ((/a \cup /a//*) \cup /a) - (/a//b// * \cup /a/b//*)$

(empty)

$V_2 : (/a//b - /a/b) \cap ((/a \cup /a//*) - /a) - (/a//b// * \cup /a/b//*)$

$V_3 : (/a//b - /a/b) \cap (/a - (/a \cup /a//*)) - (/a//b//* \cup /a/b//*)$(empty)

$V_4 : (/a//b - /a/b) \cap ((/a \cup /a//*) \cap /a) - (/a//b// * \cup /a/b//*)$

$V_5 : (/a/b - /a//b) - (((/a \cup /a//*) \cup /a) - (/a//b//* \cup /a/b//*)$(empty)

$V_6 : (/a/b - /a//b) \cap ((/a \cup /a//*) - /a) - (/a//b//* \cup /a/b//*)$(empty)

$V_7 : (/a/b - /a//b) \cap (/a - (/a \cup /a//*)) - (/a//b//* \cup /a/b//*)$(empty)

$V_8 : (/a/b - /a//b) \cap ((/a \cup /a//*) \cap /a) - (/a//b//* \cup /a/b//*)$(empty)

$V_9 : (/a//b \cap /a/b) - ((/a \cup /a//*) \cup /a) - (/a//b//* \cup /a/b//*)$(empty)

$V_{10} : (/a//b \cap /a/b) \cap ((/a \cup /a//*) - /a) - (/a//b// * \cup /a/b//*)$

$V_{11} : (/a//b \cap /a/b) \cap (/a - (/a \cup /a//*)) - (/a//b//* \cup /a/b//*)$(empty)

$V_{12} : (/a//b \cap /a/b) \cap ((/a \cup /a//*) \cap /a) - (/a//b// * \cup /a/b//*)$

and 28 triplets:

$Q_1 \leftarrow (V_1, /Ans/*)$

$Q_1 \leftarrow (V_2, /Ans/*)$

$Q_1 \leftarrow (V_2, /Ans/ * //b)$

$Q_2 \leftarrow (V_3, /Ans/*/b)$

$Q_1 \leftarrow (V_4, /Ans/*)$

.

.

.

$Q_1 \leftarrow (V_1 2, /Ans/*//b)$

$Q_2 \leftarrow (V_1 2, /Ans/*/b)$

As we can see, V_1, V_3, V_5, V_6, V_7, V_8, V_9 and V_3 are empty views. When given n recursive queries whose total length is l, this algorithm produces $(2_n - 1) * 2_{l-n}$ views, among which many are empty views. Some technique was adopt to eliminate empty views before sending them to the server, however, our implementation does not include this step as it will not affect the correctness.

3.5 Limitation

If we only measure the size of the answer sets sent from the server to the client, Tajima's algorithm is optimal as the view set generated is guaranteed to be disjoint and hence minimal. However, the views being submitted to the server are often more complicated than the original input queries, especially for the recursive queries. When a set of recursive queries are processed,

potentially the number of views grow exponentially in the total number of location steps of the input queries, which results in a high computation cost of the XPath processor at the server side. Moreover, the evaluation of $-Q_i//*$ at the end of a view will cause a very high computation cost itself. Once an input query set includes one query with //, the whole query set would be treated as recursive queries. In this case a large number of views are submitted and $-Q_i//*$s are evaluated for every single view, even if all the other queries are non-recursive and the query with // itself actually address a part of XML data without recursion. Obviously this "blindness" causes a big waste and makes the algorithm inefficient for recursive queries with respect to the computation cost.

To solve this problem, we propose a new approach that is independent of the input query type. The details are given in the next chapter.

4 Server-based Approach

In this chapter, we present a new approach to solve the redundancy problem for multi-XPath query processing. Since the redundancy elimination is done at the server side, we call it server-based approach. Unlike the client-based approach proposed by [27], the server-based approach is independent to the structure of the input XPath query, and therefore we do not have different methods for non-recursive and recursive queries. The basic idea of our work is to replace the redundant data with pointers before sending the query result back to the client. Two different methods are presented for pointer insertion: the *Embedded Pointer* approach and the *Separate Pointer* approach. The tradeoff between theses two methods and the client-based approach is also discussed.

4.1 Overview

As described in Chapter 3, the client-based approach works like a proxy server which resides at the client side. It firstly breaks the input queries into a set of minimal views for the server and then compute the real answer out of the answers to the views for the client. On the other hand, our server-side approach pushes the main work to the server side. The server receives the

original input queries and output an optimized answer sets with redundant data being replaced by smaller pointers, whereas the client only needs to do some simple I/O operations to retrieve the real data represented by the pointers.

At the server side, we have a enhanced query processor to evaluate the input queries and find out the redundancy between answer sets in the meantime. Afterwards a *pointer generator* is executed to replace redundant data blocks with pointers while writing answer sets into files. When the client receives a set of optimized answer files from the server, a *pointer interpreter* is executed to find out all pointers in each answer file and retrieve the original data block represented by those pointers. The pointer interpreter can be considered as a reversion of the pointer generator. It replaces pointers with original data block.

The core of our work is about pointer insertion. Let us see with a simple example how the pointers work to eliminate redundancies. Given an XML database T as shown in *Figure 3* that resides at the server side, suppose two simple recursive queries are submitted to the server:

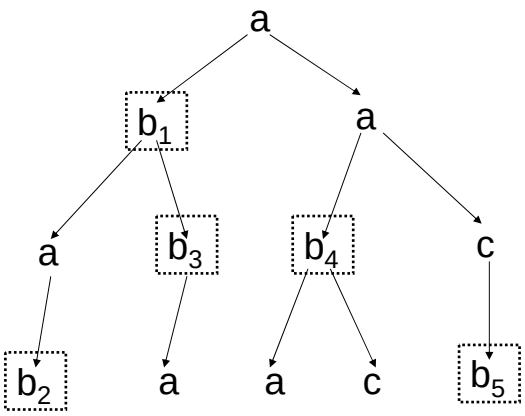

Figure 5: Tree structure of XML database at the server side

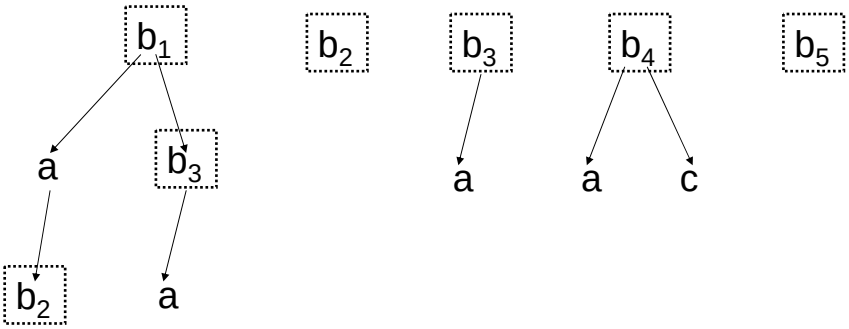

Figure 6: Answer set to Q1 in tree structure

Example 1 $\begin{cases} Q_1, & \text{/a//b;} \\ Q_2, & \text{//a/b.} \end{cases}$

Obviously every sub-tree rooted by node b in T can be a possible answer to the given queries. For the convenience of discussion, we label each node b in T with a unique id number. In case of a dummy XPath processor, the server would send two answer sets back to the client, as shown in Figure 6 and Figure 7. We can see there exists self-redundancy in both answer sets to Q_1 and Q_2 as the subtrees rooted by b_2 and b_3 appear more than once in

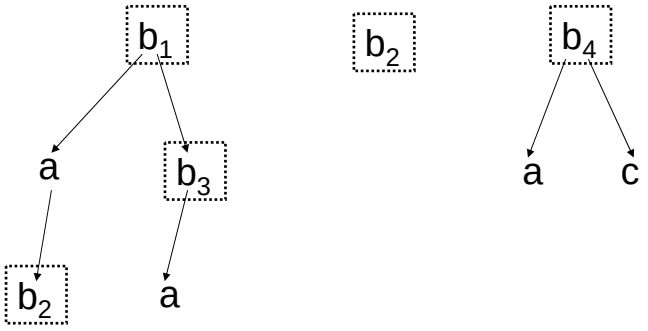

Figure 7: Answer set to Q2 in tree structure

one answer set, whereas the subtrees rooted by b_1, b_1 and b_4 are contained in both answer sets to Q1 and Q2. To eliminate these kinds of redundancies, our enhanced query processor produces an answer set with redundant subtrees being replaced by pointers. We can safely say that pointers is most likely much smaller than real data, by this assertion the size of the refined answer sets is dramatically reduced after some large XML fragments are replaced by smaller pointers.

In this thesis we proposed two different methods for pointer insertion as shown in Figure 8 and Figure 9 respectively, where circles labelled by P_ijs represent pointers. The *Embedded Pointer* method mixes real data and pointers in answer sets. In the answer set to Q_1, the subtrees rooted by b_2 and b_3 were replaced with pointers referring to b_2 and b_3 respectively, in the subtree rooted at b_1, whereas the answer set to Q_2 only contains three pointers referring to the subtrees rooted by b_1, b_2 and b_4 in the answer set

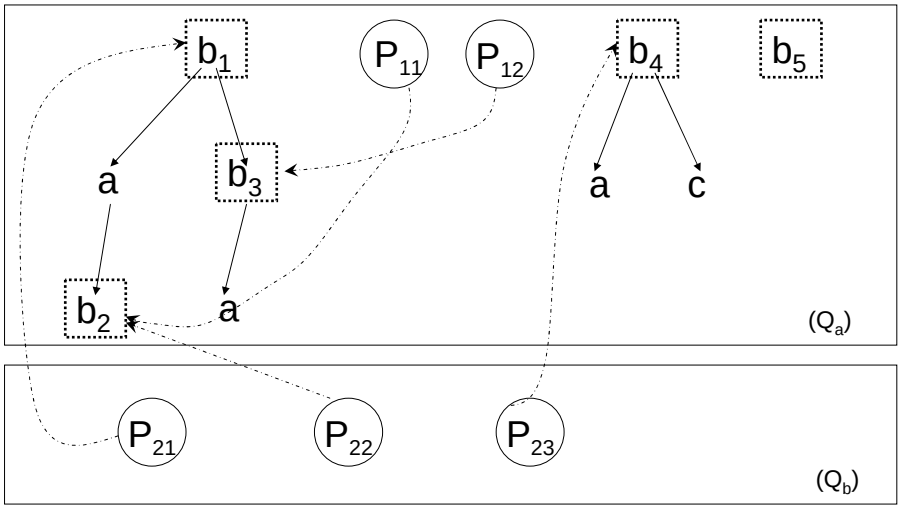

Figure 8: Optimized answer set produced by *Embedded Pointer*

to Q_a. The *Separate Pointer* method, on the other hand, stores XML data and pointers separately. It produces a text file containing all answers to both queries and a pointer set for each query containing all pointers referring to the appropriate part of the text file. The answer sets produced by both method contains no redundancy as every node appears only once. The first method produces fewer pointers whilst the second method is more straight forward and less expensive in computation. However, the details will be presented in the following subsections.

In this thesis, we make a assumptions about input XPath query language. We assume the input XPath queries are all structural queries, as the atomic answers to a aggregate queries will not cause any redundancy. However, the aggregate functions can still be used as predicates in filter expressions,

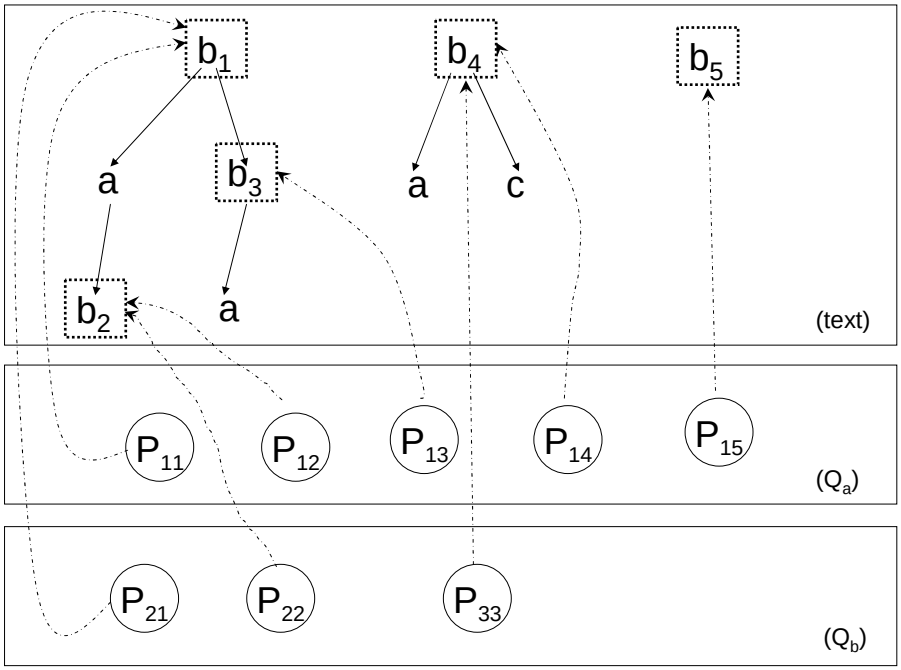

Figure 9: Optimized answer set produced by *Separate Pointer*

though this kind of predicates will not alter the nested structure of final answers and therefore is not going to be discussed in this thesis.

4.2 Enhanced Query Processor

Our enhanced query processor at the server side is based on an ordinary XPath query processor, which outputs a list of XML nodes for an input XPath query. To avoid redundancy in query answers, a *Node Table* is created to keep track of every distinct answer node and the queries it matches. A list of query-id for each distinctive node is stored in *Node Table*.

Given a set of input XPath queries, the enhanced processor works as follows. For each answer node, it first checks its existence in the *Node Table*. If the *Node Table* does not contain an entry for this node, create a new one for it, otherwise find the existing respective entry and update its query-id list. The pseudocode representation of this procedure is given in Algorithm 1.

Algorithm 1 Enhanced Processor

```
1: create an empty Node Table
2: for each input query $q do
3:    evaluate $q
4:    for each node $n in the answer list of $q do
5:       look $n up in the Node Table
6:       if there is an entry corresponding to $n then
7:          fetch this entry and add $q id to its query-id list
8:       else
9:          create a new entry with $n mapping to an empty list l
10:         add $q id to $l
11:      end if
12:   end for
13: end for
```

After the whole set of input queries have been processed, an *Answer Table* is derived from the *Node Table*. It keeps a list of all answer nodes to each input query. Figure 10 shows the *Node Table* and *Answer Table* for Example 1.

With *Node Table* and *Answer Table*, we have already got all the answer sets which share some nodes between each other. The server then writes

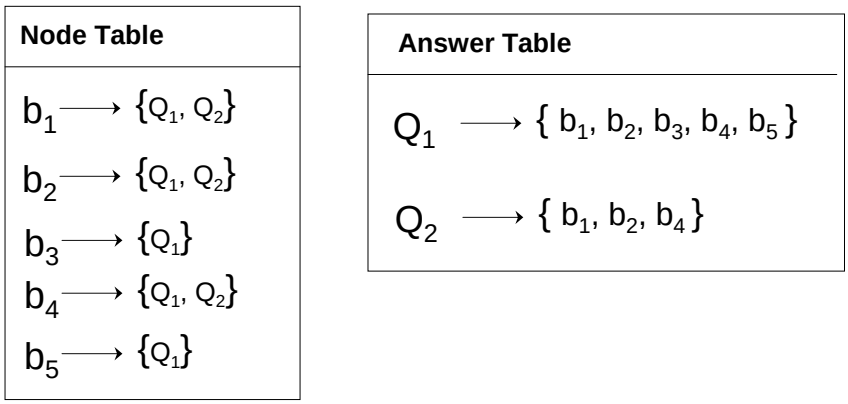

Figure 10: Node Table and Answer Table for Example 1

these answer sets into files before transmitting them to the client. During the process of data writing, numerous pointers are brought to replace the data blocks that have been written before. After the client receives the optimized answers, it interprets the pointers to get the real answer sets. Basically a pointer contains necessary information for the client to retrieve the original data block. We use the position of the first character and the total number of characters to address a data block in a specific text file. Moreover, a file-id is also needed for *Embedded Pointer* as the referred data block might be included in any answer file. For example, a pointer "3/1/48" refers to a data block of forty-eight characters starting from the first character in the answer file corresponding to Q_3.

4.3 Embedded Pointer Approach

As illustrated in Figure 8, the *Embedded Pointer* approach embeds pointers into the original answer sets. The basic idea is rather intuitive: replace the subtrees that are contained in other answers with pointers. When the client submits N input XPath queries, the server sends back exactly N answer files with each file corresponding to one input query.

An answer file produced by this method is a mixture of XML data and pointers, therefore every pointer is enclosed by special tags <AnsPtr>< /AnsPtr>, which then become reserved words. We assume the source XML database does not contain this kind of tags. Since a pointer can refer to an XML data block in any specific answer file, a file-id is included besides the starting offset and the number of character in this data block as follows:

<AnsPtr>file-id / starting offset / size </AnsPtr>

Let us see with our simple example how the server generates optimized answer files and how the client retrieves the real answer sets out of them.

4.3.1 Server Side

At the server side, we have a pointer generator to generate answer files based on *Answer Table* and *Node Table* produced by the enhanced query processor.

For every entry in *Answer Table*, the pointer generator creates a file representing the answer to the corresponding query. Before writing out the file, it first sorts the node list by document order of the XML source. By performing this step, the document order of the final answer sets are preserved. Then it writes the node list into the answer file. For each node, it first checks whether it is included in a list which has been processed previously and writes it if it is not found. In the meantime, the starting position and the size of this text block as well as the current query-id are recorded to make a pointer that could be used to address the subtree rooted by this node. Otherwise, it simply writes the pointer recorded in previous operation.

Because of the nested structure of XML document, a recursive method is created to print every node into answer file. It performs a depth-first traversal of the XML subtree rooted by the current node. For every node visited, it first checks whether it has a corresponding entry in *Node Table*. If not, it simply outputs this node; otherwise, it is an answer node that matches some input query and may result in redundancy. The method then records the necessary pointer information in *Node Table* while writing it into its answer file. In case the pointer record has already existed, it writes the pointer instead, since the subtree rooted by this node is included in the answer set of

some query that has already been processed.

Algorithm 2 is the pseudo-code representation of how the pointer generator works.

Algorithm 2 Pointer Generator 1

```
1: for each entry in Answer Table do
2:    create an answer file $f
3:    sort node list $l in document order
4:    for each node n in $l do
5:       fetch the corresponding entry of $n in the Node Table
6:       if there is a pointer $p recorded for $n  then
7:          write $p into $f
8:       else
9:          start := current position in file stream
10:         Print-Subtree($n, $f, current query-id)
11:         size := number of characters being written
12:         add pointer "query-id/start/size" into Node Table
13:      end if
14:   end for
15: end for
```

It is easy to prove that the answer files generated by this algorithm contains no redundancy. There is not a single element appearing in more than two answer sets, as every answer node is replaced by a pointer in case it is already included in some answer file. Moreover, the *Print-Subtree* procedure checks the whole subtree to decide whether a subelement is an answer node itself. This procedure guarantees that no data block could be part of other data blocks in the answer files, in other words, there is no redundancy caused

Procedure 3 Print-Subtree(root, answer file, query-id)

```
1: if root is already a leaf node then
2:    write root into answer file
3: else
4:    for each child node $n of root do
5:       loop $n up in the Node Table
6:       if there is no corresponding entry for $n then
7:          Print-Subtree($n, answer file, query-id)
8:       else
9:          fetch the corresponding entry of $n in Node Table
10:         if there is a pointer $p recorded for $n  then
11:            write $p into answer file
12:         else
13:            start := current position in file stream
14:            Print-Subtree($n, answer file, query-id)
15:            size := number of characters being written
16:            add pointer "query-id/start/size" into Node Table
17:         end if
18:      end if
19:   end for
20: end if
```

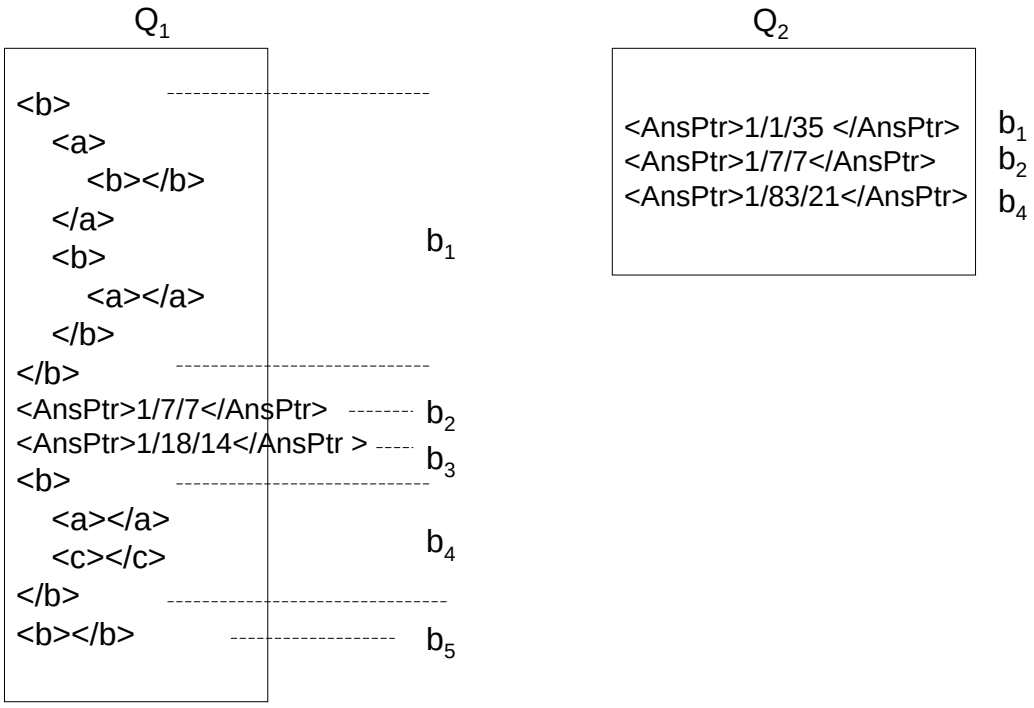

Figure 11: Answer file generated for Q_1 and Q_2 by *Embedded Pointer*

by answers appearing as substructure of the other answers. In conclusion, the answer sets are disjoint and hence optimal.

For the simple example queries Q_1 and Q_2, the enhanced processor produced a *Node Table* and an *Answer Table* as shown in Figure 10. The pointer generator then processes the entries in *Answer Table* one by one. It first fetches the sorted answer list of Q_1 which is $\{b_1, b_2, b_3, b_4, b_5\}$ and starts with b_1:

b_1**:** As b_1 is also an answer node of Q_2 according to *Node Table*, a pointer "1/1/35" is recorded for future reference after it is printed by the Print-

Subtree procedure.

In this pointer, the first token "1" stands for Q_1, which means the data block represented by this pointer resides in the answer file of Q_1; whereas the second token "1" is the starting offset of this data block and the third token "35" is the number of characters in this data block, which means the subtree referred by this pointer contains 35 characters starting from first character in the answer file corresponding to Q_1. For the convenience of discussion, only visible characters are counted in this paper. However, some overhead like newline character is also considered in real implementation.

When the Print-Subtree procedure is invoked to print the subtree rooted by b_1, two pointers "1/7/7" and "1/18/14" are taken for b_2 and $b3$ respectively, as they are both contained in *Node Table*.

b_2: According to *Node Table*, the pointer "1/7/7" is available for b_2, so it is printed instead of the real data.

b_3: Pointer "1/18/14" is written into the answer file, same as b_2.

b_4: According to *Node Table*, b_4 has not been processed but it answers both Q_1 and Q_2, therefore it is written into the answer file and a pointer "1/83/21" is taken as well.

b_5**:** According to *Node Table*, b_5 only answers one input query, therefore the generater simply prints it without recording a pointer.

As for answer node list of Q_2, three pointers "1/1/35", "1/7/7" and "1/63/21" are printed since subtrees rooted by b_1, b_2 and b_4 have all been included in Q_1's answer file. The two answer files generated are shown in Figure 11.

4.3.2 Client Side

At the client side, we have a pointer interpreter to retrieve the real answers out of the the optimized answer files received from the server. The pointer interpreter is like a reversion of the pointer generator: it replaces pointers with original data block they refer to. The basic idea of the pointer interpreter is rather intuitive. It reads every answer file line by line, every line enclosed by reserved tags $< AnsPtr >$ and $< /AnsPtr >$ is interpreted as a pointer. It then retrieves the data blocks represented by this pointer. For example, when line "7/7" is processed, it reads seven characters starting from the seventh character in the data file, which is "$< b >< /b >$".

However, in some cases this step becomes non-trivial as the data block represented by a pointer may contain pointers when subelements are processed before their ancestors. This only occurs in cross-file reference, because

ancestors are guaranteed to be processed before their descendants as the node list is sorted by document order. For example, consider three queries Q_a, Q_b and Q_c to be processed in this order. Suppose Q_a's answer file contains an element a, Q_b's contains an element b and Q_c's contains an element c, where a is a subelement of b and b is a subelement of c. The pointer generator therefore inserts pointers referring to a and b when it processes element b and c respectively. When the pointer interpreter interprets the pointer referring to b in Q_c's answer file, it retrieves a data block containing a pointer referring to element a in Q_a's answer file. A recursive method Retrieve-Data is then created to solve this problem. It interprets pointers recursively until the source data retrieved does not contain any pointer.

The pseudo-code representation is given in Algorithm 4.

Algorithm 4 Pointer Interpreter 1

```
1: for each answer file $afile do
2:    for each line $l in $afile do
3:       if $l is enclosed by < AnsPtr >< /AnsPtr > then
4:          Retrieve-Data($l, $afile)
5:       else
6:          write $l into $afile
7:       end if
8:    end for
9: end for
```

Procedure 5 Retrieve-Data(pointer, answer file)

1: interpret the pointer and get $\$file-id$, $\$starting-offset$ and $\$size$
2: retrieve $\$size$ characters starting from $\$starting-offset$ the answer file identified by $file-id$
3: **for** each line $\$l$ in this data block **do**
4: **if** $\$l$ is enclosed by $< P >< /P >$ **then**
5: Retrieve-Data($\$l$, answer file)
6: **else**
7: write $\$l$ into answer file
8: **end if**
9: **end for**

4.4 Separate Pointer Approach

As illustrated in Figure 9, the *Separate Pointer* approach stores XML data and pointers separately. In particular the pointer generator writes all possible answer nodes of the whole input query set into one single text file and generates pointers referring to various part of this file. When the client submits N input Xpath queries, the server sends back $N+1$ answer files,among which there are 1 data file and N pointer files with each file corresponding to one input query.

As a pointer file produced by this method contains nothing but pointers, no tags are needed to separate pointers from data. Furthermore, it's not necessary to specify a file-id since the data block a pointer refers to always resides in one single file. Therefore a pointer file becomes a collection of lines which consists of the starting offsets and the size of the source data blocks.

$starting - offset/size$

Let us still use our simple example to show how the server generates data and pointer files as well as how the pointer interpreter at client side retrieves the real answer out of them.

4.4.1 Server Side

Although the basic idea about replacing redundant data with pointers is more or less the same, the pointer generator of *Separate Pointer* method works a bit differently from *Embedded Pointer* method. It generates the data file based on *Node Table* and pointer files based on *Answer Table*.

Firstly it sorts the *Node Table* by document order of the XML source. By performing this step, the document order of the final answer sets are preserved. Furthermore, ancestors are guaranteed to be written prior to their descendants. Therefore, for an element to be processed, the generator either writes the whole subtree or does nothing when it is already contained in an element which is processed before.

Then it creates a text file F and writes answer nodes into it. For each node in the sorted *Node Table*, it first checks if there is a corresponding

pointer available for this node. If not, it writes the subtree rooted by this node into F and records a pointer consists of the starting position and the number of characters of this data block. Otherwise this node is skipped because it is already included in F when an ancestor of this node is processed.

Similar to the *Embedded Pointer* method, a recursive method is created for subtree writing because of the nested structure of XML document. It also performs a depth-first traversal of the subtree. For every node visited it first checks *Node Table*. A node include in *Node Table* itself is an answer node to some input query, therefore a pointer is recoded when it is being written into the *F*.

Algorithm 6 is the pseudo-code representation of how this pointer generator works.

For the queries Q_1 and Q_2 in Example 1, the enhanced processor produced a *Node Table* and an *Answer Table* as shown in Figure 10. The pointer generator then process the *Node Table* in document order:

b_1**:** A pointer "1/35" is recorded for future reference after it is printed by the Print-Subtree procedure. When the Print-Subtree procedure is invoked

Algorithm 6 Pointer Generator 2

```
1: sort Node Table in document order
2: create a file $data
3: for each node $n in Node Table do
4:    if there is a pointer recorded for $n then
5:       do nothing
6:    else
7:       start := current position in file stream
8:       Print-Subtree($n, $data)
9:       size := number of characters being written
10:      add pointer "start/size" into Node Table
11:   end if
12: end for
13: for each entry in Answer Table do
14:   create a file $F
15:   for each node in the node list do
16:      fetch the corresponding entry from the Node Table
17:      write the recorded pointer into $F
18:   end for
19: end for
```

Procedure 7 Print-Subtree(root, data file)

```
1: if root is already a leaf node then
2:    write root into data file
3: else
4:    for each child node $n of root do
5:       look $n up in the Node Table
6:       if there is no corresponding entry for $n then
7:          Print-Subtree($n, data file)
8:       else
9:          start := current position in file stream
10:         Print-Subtree($n, data file)
11:         size := number of characters being written
12:         add pointer "start/size" into Node Table
13:      end if
14:   end for
15: end if
```

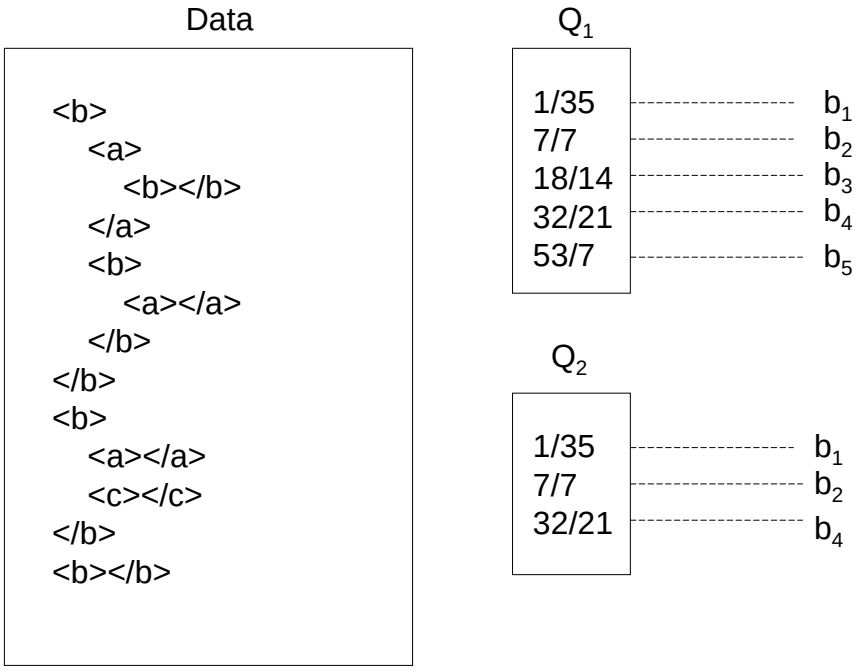

Figure 12: Data and pointer files generated for Q_1 and Q_2 by *Separate Pointer*

to print the subtree rooted by b_1, two pointers "7/7" and "18/14" are taken for b_2 and b_3 respectively, as they are both contained in *Node Table*.

b_2: It is skipped, as there is a pointer available in *Node Table*.

b_3: It is skipped like b_2.

b_4: A pointer "32/21" is recorded for future reference after it is printed.

b_5: Similar to b_4, a pointer "53/7" is recorded for future reference.

After the whole *Node Table* has been processed, the pointer generator creates a pointer file for each entry in *Answer Table*. For every node in the

node list, it writes its pointer into the corresponding pointer file. The text file and two pointer files generated are shown in Figure 12.

4.4.2 Client Side

The pointer generator of *Separate Pointer* is rather simple. It reads a pointer file line by line, interprets the pointer and retrieves the text block it refers to from the text file. The details of this procedure is presented in Algorithm 8.

Algorithm 8 Pointer Interpreter 2

```
1: for each pointer file $Pfile do
2:     create an answer file $Afile
3:     for each line "start/size" in $Pfile do
4:         read $size characters starting from $start from data file, write them
           into $Afile
5:     end for
6: end for
```

4.5 Discussion

The *Embedded Pointer* approach only generates pointers for redundant elements which has already been processed, whereas the *Separate Pointer* approach generates pointers for every element in the whole answer set. Obviously, *Separate Pointer* produces more pointers than *Embedded Pointer* does. In particular, if there are N distinct answer nodes for an input query set, *Separate Pointer* would produce N more pointers than *Embedded Pointer*

does. Therefore *Separate Pointer* is only applicable when there is a reasonable amount of overlapping between input queries, otherwise the percentage of unnecessary pointers would be too high comparing to *Embedded Pointer*. In case of a disjoint input query set, *Embedded Pointer* would produce the same answer set as a dummy XPath processor does, whereas *Separate Pointer* would still create a set of pointer files and the pointer interpreter at the client side is still needed. Generally *Embedded Pointer* has better performance with respect to communication cost, although *Separate Pointer* could possibly generates files of smaller size when there is enough overlapping among the input queries, since its pointers are shorter than *Embedded Pointer*'s.

However, *Separate Pointer* is overall less expensive than *Embedded Pointer* when it comes to the computational cost. It is because its pointer generator has less table lookup at the server side and its pointer interpreter needs less I/O operation at the client side.

In comparison to the client-based approach, our server-based approch has its pros and cons. Basically the files transferred from our server include more overhead, as their files only contain the real data, without any pointer. But the views submitted by their client are more complicated than the original input queries and the number of views can be exponential to the total length

of original input queries. Therefore this client-based approach would become inefficient or even impracticable when there are a large number of input queries to be processed simultaneously. As for the computation cost, their evaluation of recursive queries could be very expensive because of the expression $-Q_i//*$, especially when there are a large number of input queries. In this case, the post-processing of optimized answers also becomes expensive because of the large number of query evaluations at the client side.

5 Experimental Results

To validate the effectiveness of the proposed server-based approach, we have implemented both the *Embedded Pointer* approach and the *Separate Pointer* approach as well as Tajima's algorithm in Java based on a DOM-based XPath processor Saxon [25], which outputs a list of XML nodes for an input XPath query. Our data is generated by *xmlgen* [26], which is a benchmark data generator based on XMark. Our test queries are generated manually based on various experimental purpose.

In order to study the tradeoff between the space and time efficiency, we use both communication cost and computation cost as performance metric. The communication cost is measured in bytes, including the size of input queries or views submitted by the client and the size of answers returned by the server. The computation cost is measured in milliseconds, including the execution time at both the server side and the client side. We also choose five typical network speeds to compute the total processing time.

Firstly, we ran experiments on query sets with various degree of overlapping to see whether the performance of the *Embedded Pointer* approach and the *Separate Pointer* approach varies as discussed in Section 4.5. Then we

tested sets of typical queries to compare the performance of our server-based approaches and Tajima's client-based approach. Moreover, we also vary the size of XML document to see how these approaches behave. Finally we compare the effectiveness of all three approaches in networks of different speeds.

5.1 Embedded Pointer vs Separate Pointer

In order to compare the performance of *Embedded Pointer* approach and *Separate Pointer* approach, we tested three sets of typical XPath queries with different degree of overlapping as shown in Table 1. Here we use range predicates to control the degree of overlapping. These queries ask for the profile of some people whose ages are within a certain range. As we can see, the queries in Set A are disjoint, while the queries in Set B have a little overlapping and the queries in Set C immensely overlapped with one another.

Table 2 shows the communication cost measured in bytes, including the size of input queries submitted by the client and the size of answers returned by the server. In order to give a clearer picture, the percentage of result size reduction is shown in Figure 13. As we can see, the communication cost reduction is more when there is more overlapping between input queries.

	Queries	Characteristic
Set A	/site/people/person/profile [age<21] /site/people/person/profile [age>21 and age<25] /site/people/person/profile [age>25 and age*a* <40] /site/people/person/profile [age>40]	Disjoint queries.
Set B	/site/people/person/profile[age<21] /site/people/person/profile[age>18 and age<25] /site/people/person/profile[age>22 and age<40] /site/people/person/profile[age>35]	Queries with little overlapping.
Set C	/site/people/person/profile[age>1] /site/people/person/profile[age>18] /site/people/person/profile[age>21] /site/people/person/profile[age<60]	Queries which are immensely overlapped

Table 1: Query Sets with Different Degree of Overlapping

	Set A	Set B	Set C
Direct Approach	71,270	92,804	264,389
Embedded Pointer	71,270	79,313	108,543
Separate Pointer	78,096	85,583	102,053

Table 2: Comparison of Communication Cost (byte)

For a disjoint query set like Set A, *Embedded Pointer* generates the real answer directly, whereas *Separate Pointer* creates 9.57% overhead because of unnecessary pointers. For a query set with reasonable sharing like Set B, *Embedded Pointer* has greater reduction than *Separate Pointer* as it creates less pointers. However, when there is enough overlapping among queries in a query set like Set C, *Separate Pointer* has better performance with respect to the communication cost. This is because *Separate Pointer* generates smaller pointers, while the size of a single pointer becomes critical when there are a huge number of pointers.

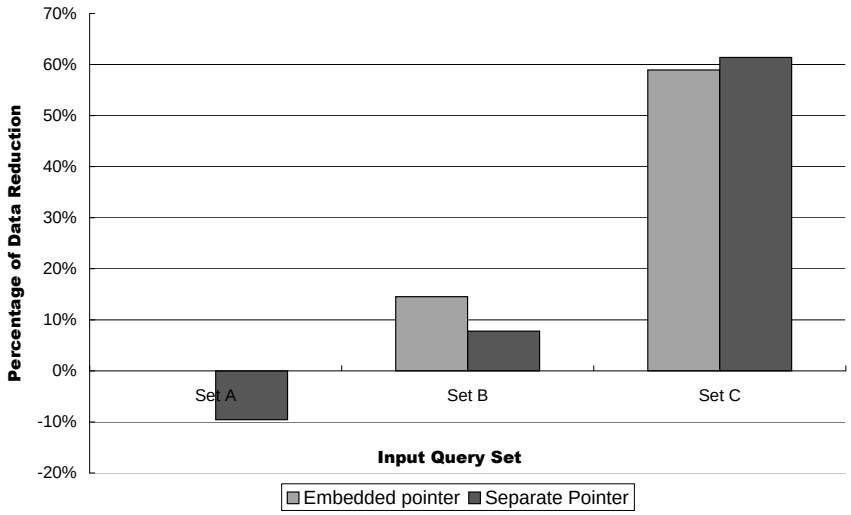

Figure 13: Percentage of Data Reduction

	Embedded Pointer		**Separate Pointer**	
	sever side	client side	server side	client side
Set A	7,782	9	7,694	299
Set B	7,778	1,445	7,613	347
Set C	7,739	1,198	7,607	371

Table 3: Computational Cost (ms) of Two Methods

Table 3 shows the computation cost in milliseconds, which consists of the execution time of the query evaluation and pointer generation at the server side and the pointer interpretation at the client side. Obviously, *Separate Pointer* is less expensive than *Embedded Pointer* both at the server side and at the client side, except for the disjoint query set where unnecessary pointers are processed.

In summary, as discussed in Section 4.5., the *Embedded Pointer* approach has a better performance with respect to the communication cost but is more

expensive when it comes to the computational cost. The *Separate Pointer* approach performs best when there is a certain amount of overlapping among input queries. The different behavior of these two approaches will be experimented further in next section.

5.2 Server-based Approach vs Client-based Approach

To compare our server-based approaches with Tajima's client-based approach, we tested three input query sets of different types, as shown in Table 4. Set 1 contains redundancy caused by common elements among different answer sets. Since Tajima's algorithm does not support range queries, we use optional branch to create the overlapping. Set 2, on the other hand, contains redundancy caused by some elements appearing as the subelements of other elements. Both Set 1 and Set 2 only contain non-recursive queries and Tajima's algorithm for non-recursive queries is applied, while Set 3 consists of queries with // and Tajima's algorithm for the general case is applied. Moreover, we tested each query set on three XML documents at sizes 58MB, 116MB and 175MB.

Table 5 shows the communication cost measured in bytes, which includes the size of input queries or views submitted by the client and the size of answers returned by the server. To make a clearer comparison, three column

	Queries	Characteristic
Set 1	/site/people/person[phone] /site/people/person[address] /site/people/person[homepage] /site/people/person[watches]	Some elements are shared by different answer sets.
Set 2	/site/regions/*/item /site/regions/namerica /site/regions/asia/item/description /site/regions/europe/item/shipping	Some elements appear as subelements of other elements.
Set 3	/site/regions//item /site//description /site//categories //name	Recursive queries.

Table 4: Query Sets with different Characteristic

charts are shown in Figures 14, 15 and 16 for each of the query sets.

It is obvious to see that the communication cost is substantially reduced for different types of queries, and the redundancy caused by elements sharing and subelements are both eliminated. For the non-recursive queries, our server-based approach generates larger query results because of the existence of pointers. However, when Set 3 is processed on source data of 58MB, the communication cost under Tajima's algorithm is even greater than the one under the Embedded Pointer approach because of the large number of views submitted by the client side. When a set of recursive queries are processed, the number of views generated by Tajima's set-operation based algorithm grows exponentially in the total length of queries. In particular, 480 views are generated for query set 3, which is 156,912 bytes. This overhead is more

significant when the size of database is relatively small.

The comparison of computation cost in milliseconds is shown in Table 6, where the execution time of Tajima's approach at the client side includes both the pre-processing step and post-processing step. From the computation cost comparison in Table 6, we can see that the performance of both *Embedded Pointer* and *Separate Pointer* are rather consistent for both non-recursive and recursive queries, since our design is independent of the type of input queries. On the other hand, Tajima's algorithm works well for non-recursive queries but is very inefficient for recursive queries. For Set 1 and Set 2 of non-recursive queries, Tajima's algorithm is less expensive at the server side because they do not have a pointer processing step after query evaluation. It is also interesting to see that Tajima's algorithm is even less expensive than the direct approach at the server side when processing Set 2. This is because only two views ($/site/regions/\overline{\{namerica\}}/item$ and $/site/regions/namerica$) are sent to the server instead of the original four queries. However, this advantage is overshadowed by the relatively expensive query evaluation in the post-processing phase at the client side. As for Set 3 of recursive queries, the computation cost of Tajima's algorithm is amazingly high because of the large number of views and the $-Q_i//*$ to be evaluated. It is even more obvious when a larger document is tested, because of the

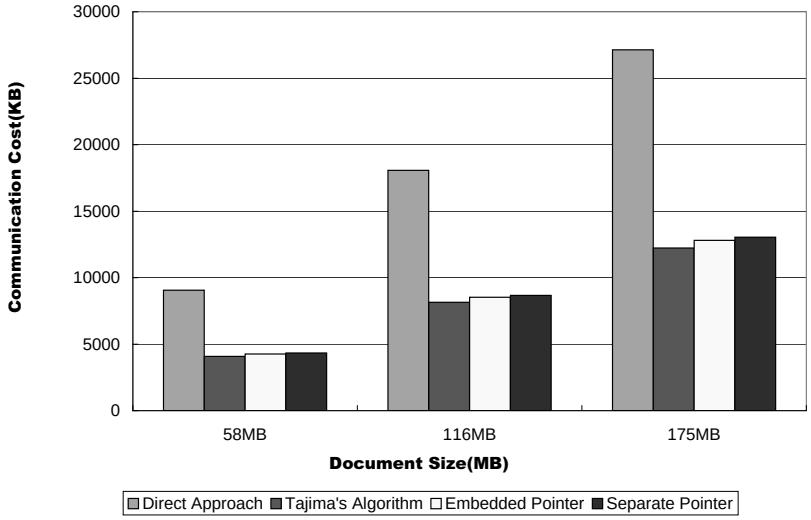

Figure 14: Comparison of Communication Cost (byte) for Processing Set 1

expensive evaluation of $-Q_i//*$ on large document.

In short, our server-based approach is consistent for both non-recursive and recursive queries. Tajima's client-based approach works well for non-recursive queries but inefficient in recursive query processing.

In order to test the effectiveness of both the server-based client-based approach comparing to the direct approach, we compute the processing time of Set 1 over different networks. We have examined five networks as follows, low speed networks like slow modem (28.8Kbps) and fast modem (56.6Kbps); medium speed network like dual ISDN (128Kbps); relatively fast network like DSL cable (384Kbps); and fast network like T1 (1.5Mbps). The total pro-

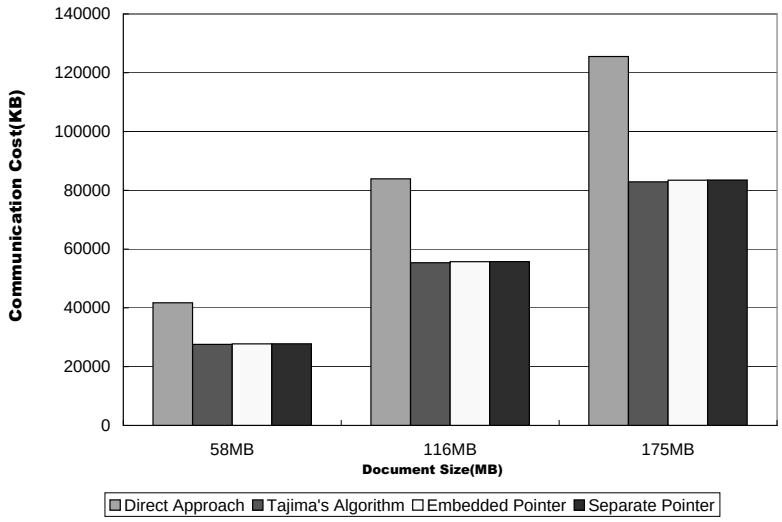

Figure 15: Comparison of Communication Cost (byte) for Processing Set 2

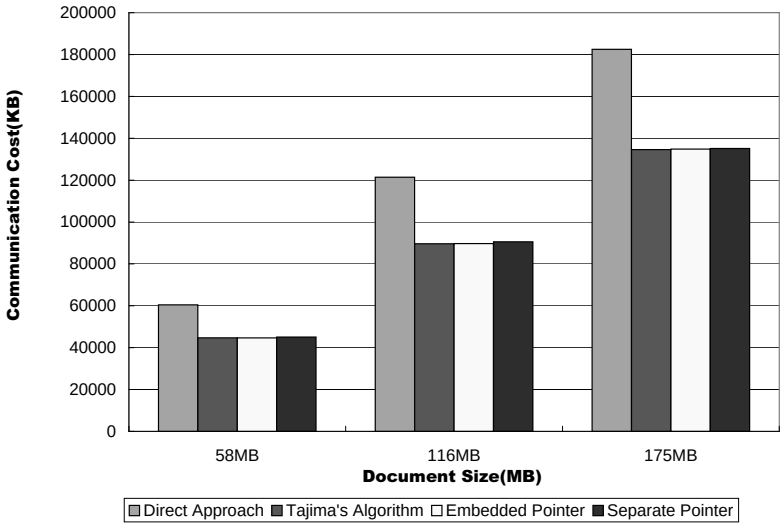

Figure 16: Comparison of Communication Cost (byte) for Processing Set 3

	Set 1		
	58MB	116MB	175MB
Direct Approach	9,060,217	18,077,194	27,137,490
Tajima's Algorithm	4,082,250	8,149,928	12,232,241
Embedded Pointer	4,262,930	8,525,598	12,808,835
Separate Pointer	4,335,425	8,669,371	13,045,061
	Set 2		
	58MB	116MB	175MB
Direct Approach	41,731,967	83,920,409	125,529,645
Tajima's Algorithm	27,592,555	55,366,052	82,906,245
Embedded Pointer	27,764,514	55,717,735	83,434,303
Separate Pointer	27,780,551	55,758,554	83,499,980
	Set 3		
	58MB	116MB	175MB
Direct Approach	60,442,963	121,430,774	182,522,574
Tajima's Algorithm	44,697,589	89,623,823	134,615,299
Embedded Pointer	44,658,172	89,721,050	134,849,809
Separate Pointer	45,070,992	90,553822	135,148,057

Table 5: Comparison of Communication Cost (byte)

cessing time in seconds, including the computation cost at both sides and the transfer time between client and server, is used as performance metric. The bar charts shown in Figures 17 and 19 give an overview of the whole situation and Table 7 shows the detailed measurements, where the best performances are underlined.

It is obvious to see that both client-based and server-based approaches work well in low and medium speed networks where the transfer cost is the bottleneck, whereas the effect is less notable in faster networks as the execution time becomes the major concern in this situation. In particular,

	Set 1					
	58MB		116MB		175MB	
	Server	Client	Server	Client	Server	Client
Direct Approach	24,113	0	45,083	0	61,606	0
Tajima's Algorithm	26,904	16,733	45,534	33,500	61,595	39,183
Embedded Pointer	26,538	3,744	46,338	6,937	64,595	8,009
Separate Pointer	25,612	1,312	41,701	2,002	63,988	2,758
	Set 2					
	58MB		116MB		175MB	
	Server	Client	Server	Client	Server	Client
Direct Approach	29,369	0	53,064	0	70,572	0
Tajima's Algorithm	25,720	14,549	48,250	35,165	62,772	42,262
Embedded Pointer	30,367	8,271	56,134	15,756	76,865	22,240
Separate Pointer	29,326	3,660	35,741	8,668	75,806	13,292
	Set 3					
	58MB		116MB		175MB	
	Server	Client	Server	Client	Server	Client
Direct Approach	30.277	0	56,674	0	79,011	0
Tajima's Algorithm	660,450	2,232	1,316,649	40749	2,065,662	54,403
Embedded Pointer	37.887	12,920	81,769	25,944	118,714	26,407
Separate Pointer	37,314	6,113	81,287	15,668	117,679	19,367

Table 6: Comparison of Computation Cost (ms)

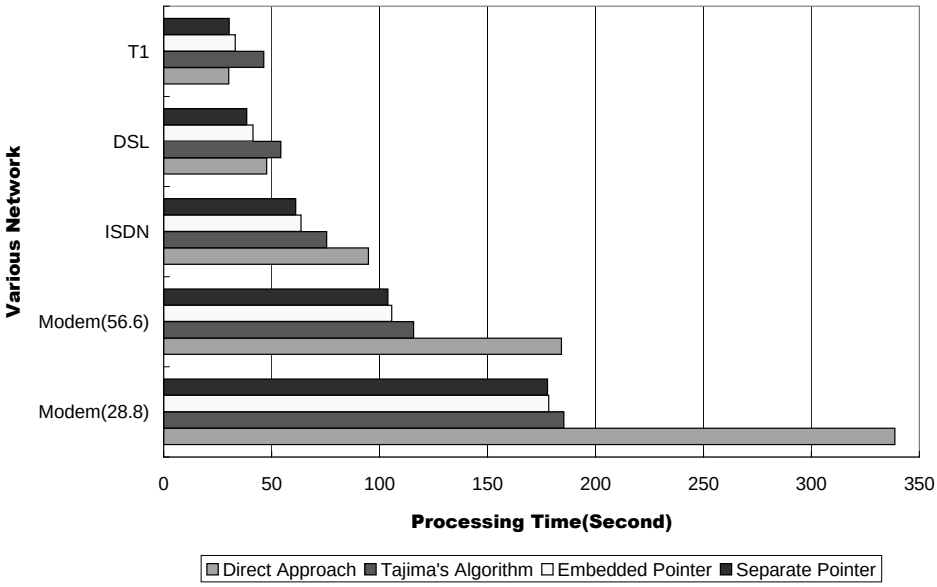

Figure 17: Processing Query Set 1 on Database of 58MB over Various Networks

	Set 1/58MB							
	Direct Approach		Tajima's Algorithm		Embedded Pointer		Separate Pointer	
	Transfer	Total	Transfer	Total	Transfer	Total	Transfer	Total
Modem(28.8)	314.57	338.69	141.71	185.35	148.02	178.3	150.54	177.76
Modem(56.6)	160.07	184.18	72.11	115.75	75.32	105.60	76.60	103.83
ISDN	70.78	94.89	31.88	75.52	33.30	63.58	33.87	61.10
DSL	23.59	47.70	10.63	54.27	11.10	41.38	11.29	38.52
T1	6.04	30.15	2.72	46.36	2.84	33.12	2.89	30.32
	Set 1/116MB							
	Direct Approach		Tajima's Algorithm		Embedded Pointer		Separate Pointer	
	Transfer	Total	Transfer	Total	Transfer	Total	Transfer	Total
Modem(28.8)	627.68	672.76	276.3	355.39	289.09	342.36	293.96	337.66
Modem(56.6)	319.39	364.47	140.61	219.65	147.10	200.37	149.58	193.283
ISDN	137.92	183.00	62.18	141.21	65.05	118.32	66.14	109.843
DSL	45.97	91.06	20.73	99.76	21.68	74.96	22.05	65.753
T1	11.49	56.58	5.18	84.22	5.42	58.69	5.51	49.212
	Set 1/175MB							
	Direct Approach		Tajima's Algorithm		Embedded Pointer		Separate Pointer	
	Transfer	Total	Transfer	Total	Transfer	Total	Transfer	Total
Modem(28.8)	942.27	1003.88	424.70	525.47	444.75	517.12	452.95	519.70
Modem(56.6)	479.46	541.07	216.10	316.88	226.30	298.67	230.48	297.224
ISDN	212.01	273.63	95.56	196.33	100.67	172.43	101.91	168.66
DSL	70.67	132.28	31.85	132.63	33.36	105.72	33.97	100.72
T1	18.09	79.70	8.15	108.93	8.54	80.90	8.70	75.44

Table 7: Performance Comparison on Various Network (second)

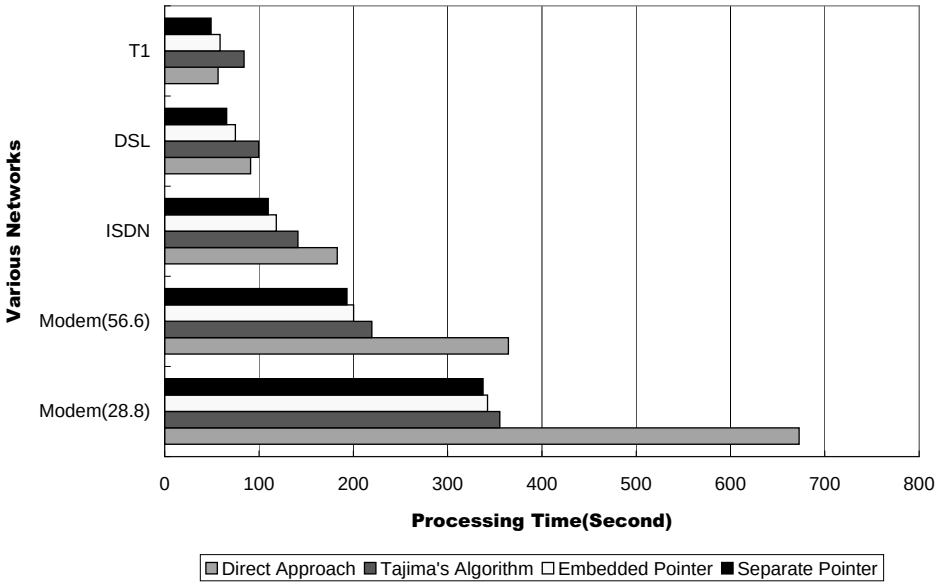

Figure 18: Processing Query Set 1 on Database of 116MB over Various Networks

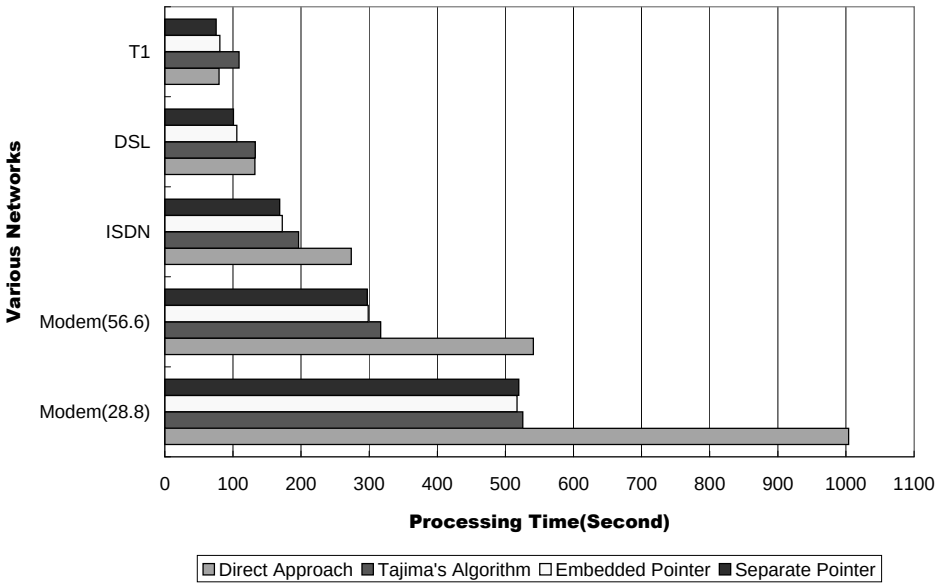

Figure 19: Processing Query Set 1 on Database of 175MB over Various Networks

although Tajima's algorithm always has the minimum transfer cost, the total costs are not very impressive because of the high computation cost of the post-processing step at the client side. On the other hand, the *Separate Pointer* approach most often has the best performance because of its relatively low computation cost. However, when the queries are processed on a document of 175MB over a very slow network of 28.8K modem, the *Embedded Pointer* approach has slightly better performance when the transfer cost becomes more significant. Similarly, the direct approach has the best performance when a document of 58MB is tested in a fast network like T1 where the computation cost is the main concern, but this slim advantage fades out when a large document of 175MB is tested as the transfer cost becomes more significant when the answer size grows.

5.3 Discussion

As all the experimental results show, both our server-based approaches and Tajima's client-based approach could substantially reduce the answer size as long as there exists redundancy among the input queries. Tajima's algorithm produces the smallest answer to be transferred, but this advantage is overshadowed by the high computation cost during the post-processing step at

	Small Data		**Large Data**	
	slightly overlapped	highly overlapped	slightly overlapped	highly overlapped
slow network	Direct Approach	Embedded Pointer	Embedded Pointer	Separate Pointer
fast network	Direct Approach	Direct Approach	Direct Approach	Separate Pointer

Table 8: Best Choice in Different Situations

the client side. It is particularly inefficient for recursive queries because of the exponentially growing number of views and the expensive evaluation of $-Q_i//*$.

Embedded Pointer approach

In short, there is no overall best approach. Table 8 roughly summaries how to choose a best approach base on different situation.

6 Conclusions

In this thesis, we have proposed and implemented an algorithm to optimize multi-XPath query processing in a client-server system with respect to the communication cost. When a client submits multiple XPath queries to the server, redundancy occurs between the answers because of the characteristics of XML and XPath: XML data has a nested structure and XPath query retrieves substructures appearing at arbitrary levels. K. Tajima et al. [27] studies this problem and proposes a client-based approach for it. However, although the proposed approach in [27] is optimal with respect to answer size transferred from server to client, it is very inefficient for recursive queries with respect to the computation cost. Therefore we propose a server-based approach which is independent of the input query type and therefore works well for both recursive and non-recursive queries.

The basic idea of the proposed server-based approach is to replace the redundant data with pointers before sending them to the client. For the pointer insertion, we designed two different methods: *Embedded Pointer* and *Separate Pointer*. As their names suggest, the *embedded pointer* approach produces a set of answer files with pointers embedded in, whereas the *separate pointer* approach produces a text file and a set of pointer files.

To validate the effectiveness of the proposed approach, we implemented the two methods and Tajima's client-based approach. Various experiments are conducted for all the three methods over different input query sets and XML data. The experimental result shows that our server-based approach can substantially reduce the size of multiple XPath query results being sent over network, which is critical in low/medium speed or high traffic network where the communication cost could easily become a bottleneck.

As the experimental results suggest, when the execution time becomes the major concern in a fast network like T1, the performance of the proposed approach could be even worse than the direct approach with respect to the total processing time. It is because the additional computation cost in pointer generating and interpreting overshadows the reduction in communication cost. In a client-server environment, the computation cost and communication cost are always a tradeoff while both [27] and our work focus on the communication and sacrifice some time efficiency. However, it would be interesting to adopt the traditional technique of multi-query optimization to reduce the execution time at the server side by exploiting the common subexpressions. It becomes an important future work.

References

[1] J. R. Alsabbagh, V. V. Raghavan: A framework for Multiple-Query Optimization. RIDE-TQP 1992: 157-162

[2] D. Calvanese, G. Giacomo, M. Lenzerini and M. Y. Vardi. Answering Regular Path Queries Using Views, In ICDE 2000: 389-398.

[3] R. Chirkova and C. Li. Materializing views with minimal size to answer queies. In *PODS*, pp. 38-48, 2003.

[4] C. Chung, J. Min, and K. Shim. APEX: An adaptive path index for XML data. In ACM SIGMOD, June 2002.

[5] F. Cooper, Neal Sample, Michael J. Franklin, Gisli Hjaltason, and Moshe Shadmon. Fast index for semistructured data. In Proc. VLDB 2001, pages 341-350, 2001.

[6] L. Chen and E. A. Rundensteiner. ACE-XQ: A CachE-aware XQuery Answering System, In *ACM SIGMOD Associated Workshop on the Web*

and Databases, Madison, Wisconsin, June 2002, pp 31–36

[7] L. Chen, E. A. Rundensteiner and S. Wang. XCache - A Semantic Caching System for XML Queries. In *ACM SIGMOD '2002 June 4-6*, Madison, Wisconsin, USA

[8] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB Conference*, pages 330–341, 1996.

[9] N. N. Dalvi, S. K. Sanghai, P. Roy and S. Sudarshan. Pipelining in multi-query optimization. In PODS. 2001.

[10] K. O'Gorman, A. El Abbadi, D. Agrawal: Multiple Query Optimization by Cache-Aware Middleware Using Query Teamwork. In *Proceedings of the 18th International Conference on Data Engineering*. 2002

[11] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata. In *Proceedings of the 9th Inter-*

national Conference on Database Theory (ICDT), pages 173–189, Siena, Italy, January 2003.

[12] G. Grahne and A. Thomo. Query Containment and Rewriting Using Views for Regular Path Queries under Constraints, In PODS 2003: 111-122.

[13] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. of 23rd Intl. Conf. on Very Large Data Bases*, August 1997.

[14] A. Y. Halevy. Answering queries using views: a survey. Technical Report, Comp. Sci. Dept., Washington Univ., 2000.

[15] J. Li, R. Chirkova and C. Li. Minimizing Data-Communication Costs by Decomposing Query Results in Client-Server Environments. Technical report, Information and Computer Science, UC Irvine, 2003.

[16] Q. Li and B. Moon, Indexing and Querying XML data for Regular Path Expressions, Proc. of VLDB, 2001.

[17] H. Mistry, P. Roy, S. Sudarshan and K. Ramamritham. Materialized view selection and maintenance using multiquery optimization, in: Proc. SIGMOD, 2001, pp. 307–318.

[18] G. Miklau and D. Suciu. Containment and equivalence for an xpath fragment. In *Proceedings of PODS*, pages 65–76, 2002.

[19] B. Mandhani and D. Suciu. Query caching and view selection for XML databases. In *proceedings of the 31st international conference*, Trondheim, Norway, 2005

[20] J. McHugh, J. Widom. Query optimization for XML. Technical report, Stanford University, 1999.

[21] Q. Ren and M. H. Dunham. Semantic Caching and Query Processing. Southern Methodist University, TR-98-CSE-04 , 1998.

[22] P. Roy, S. Seshadri, S. Sudarshan, S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. SIGMOD 2000.

[23] T. K. Sellis. Multiple-query optimization. In *ACM Transactions on Database Systems (TODS). Volume 13 , Issue 1 (March 1988) Pages: 23 - 52* 1988

[24] K. Shim, T. K. Sellis, and D. Nau, Improvements on a heuristic algorithm for multiple-query optimization, Data Knowl. Eng., vol. 12, pp. 197C222, 1994

[25] SAXON: XSLT and XQUERY processing http://www.saxonica.com

[26] A. Schmidt, et al. XMark: A benchmark for XML data management. In VLDB, pp. 974-985, 2002. http://monetdb.cwi.nl/xml/

[27] K. Tajima and Y. Fukui. Answering XPath Queries over Networks by Sending Minimal Views. In *Proceedings of VLDB*, Toronto, Canada,

Aug./Sept. 2004, pp. 48-59

[28] World Wide Web Consortium. XML Path Language (XPath) Version 1.0 http://www.w3c.org/TR/xpath.

[29] World Wide Web Consortium. Extensible Markup Language (XML) 1.1 http://www.w3.org/TR/2004/REC-xml11-20040204/

[30] H. Wang, S. Park, W. Fan, and P. S Yu. ViST: A dynamic index method for querying XML data by tree structures. In SIGMOD, 2003.

[31] W. Xu. The Framework of an XML Semantic Caching System. In *English International Workshop on the Web and Databases*. June 16-17, 2005, Baltimore, Maryland.

[32] W. Xu and Z. M. Ozsoyoglu. Rewriting XPath Queries Using Materialized Views. In *Proceedings of the 31st VLDB Conference*, Trondheim, Norway, 2005

[33] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez, Caching strategies for data-intensive Web sites, in *Proceedings of the International Conference on Very Large Data Bases* 2000.