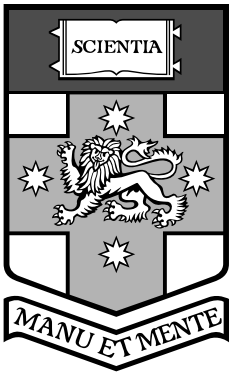
Update Synchronization for Mobile XML Data

Franky Lam Nicole Lam Raymond Wong
School of Computer Science & Engineering
University of New South Wales
Sydney, NSW 2052, Australia
wong@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-0310
June 2003

SCHOOL OF COMPUTER SCIENCE & ENGINEERING
THE UNIVERSITY OF NEW SOUTH WALES



Abstract

Many handheld applications receive data from a primary database server and operate in an intermittently connected environment these days. They maintain data consistency with data sources through sychronization. In certain applications such as sales force automation, it is highly desirable if updates on the data source can be reflected at the handheld applications immediately. This paper proposes an efficient method to synchronize XML data on multiple mobile devices. Each device retrieves and caches a local copy of data from the database source based on a regular path expression. These local copies may be overlapping or disjoint with each other. An efficient mechanism is proposed to find all the disjoint copies to avoid unnecessary synchronizations. Each update to the data source will then be checked to identify all handheld applications which are affected by the update. Communication costs can be further reduced by eliminating the forwarding of unnecessary operations to groups of mobile clients.

1 Introduction

The growing trend towards mobile computing and the increasing popularity of XML have resulted in more and more handheld applications accepting their data in XML format. Due to this, some vendors (for example, `http://www.tendara.com`) have provided handheld XML database management system for integrating enterprise applications such as sales force automation systems with a mobile workforce. Others have used XML for defining synchronization protocols between the global database servers and the mobile databases. For instance, SyncML is a proposed synchronization protocol which runs over different internet and wirelesss transports. An updategram used by Oracle and SQL Server is XML generated by agents to notify the client of changes to the data on the server, and vice versa.

Consider a database environment where an XML server database system shares portions of data (e.g., legacy data with exchange in XML format, a part of a large XML document, or a subset of document collections) with a set of intermittently connected clients. The connectivity is intermittent due to an unstable or expensive connection. Hence clients retrieve a copy of the shared data from the server and maintain it in their local database. In this paper, the retrieval language is XQL [12] extended with update operators as proposed similarly in [13, 14]. Updates made to this local database are propagated to the server database when the client connects. The data shared between the server and some Client A may also be shared with another Client B; therefore, changes to that data at Client A should be reflected at Client B. Since the clients are only intermittently connected and cannot directly send changes to other clients, the server acts as a conduit for updates by forwarding the updates to its relevant clients. In fact, the server is responsible for tracking client updates to shared data and batching those updates for dissemination to other clients which share the data.

To solve this problem, we could adopt the current approach used in most intermittently connected relational databases. In these systems, each client is treated individually such that update files are created containing updates relevant to each particular client (on a per-client basis). That is, for each client, the server prepares a client-specific update file. This is called the *client-centric* approach [9] because it aggregates database changes based on the data needed by each client. Unfortunately, the processing and sending of each client-specific file is expensive in terms of server processing and network bandwidth consumption; therefore, the server processing load is on the order of the number of clients. That is, the server incurs additional cost for each and every client, so the number of clients that can be served is limited.

[9] proposed exploiting the overlap of data shared between various clients to increase the scalability of the server. This was accomplished with data-centric processing, rather than client-centric processing, by grouping data according to how it is shared between clients. In the data-centric approach, the server creates an update file for each data group. Unlike the client-centric approach which builds an update file for each client, the data-centric approach builds update files for data groupings and requires the clients to merge the correct set of update files to retrieve the needed updates. Hence, the data-centric approach reduces the complexity of update file maintenance from the order of the number of clients to the order of the number of groups, thereby increasing the scalability of server processing.

However, as XML information is semistructured and may not have a rigid schema, the techniques proposed in [9] and also [16] cannot be applied. In this paper, we exploit this data-centric grouping idea and propose a hierarchical grouping structure based on data sharing. In particular, data sharing is determined by a client's subscription. Moreover, determination of whether an update is related to a client group becomes difficult due to the complexity of XML data and query structures.

2 Background

In this section we present the architecture of an XML-based mobile database system. We then describe how XSync uses XQL expressions as a retrieval language to specify the subset of data to be stored in the local cache of a mobile client.

2.1 An XML-based Mobile Database Architecture

Figure 1 shows the general architecture for an XML-based mobile database architecture. The back-end server, *S*, stores information which is shared between mobile devices (*A, B, C, D, E, F*). This information, which may exist in a different format, is converted to XML. Mobile devices can identify a subset of the data that is of interest by specifying a gerneral path expression for their subscription.

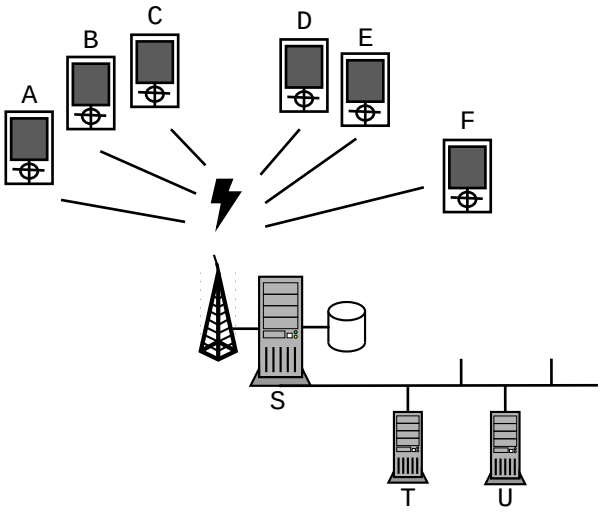


Figure 1: Architecture of an XML Based Synchronization System

2.2 XQL as an Access Language

XQL is a query language, similiar to XSL pattern syntax. It is used to address and filter the elements and text of XML documents. XML documents can be viewed as a directed acyclic graph (DAG) where every XML element can be represented as a node and an edge is represented as a relation between two nodes (Figure 2). An XQL expression consists of a set of path expressions combined using binary and set operators such as and operator and $union$ operator. A path expression contains a list of literal strings or wildcard ($*$) operators, delimited by either the child ($/$) or descendant ($//$) operator. Literal strings and wildcard operators are used to match against XML element names, while the child and descendant operators are used to match the relationship between those XML elements. Each literal string and wildcard operator can optionally contain predicates ($[]$) for filtering. A predictate contains an XQL expression with the matched element name acting as the root of the DAG.

We use an XML database management system which manages computer hardware sales force automation systems as an example throughout this paper. Different clients issue different XQL expressions to identify the subset of data they are interested in:

- Retrieve all computer systems.
  `/Product/Computers/Item`

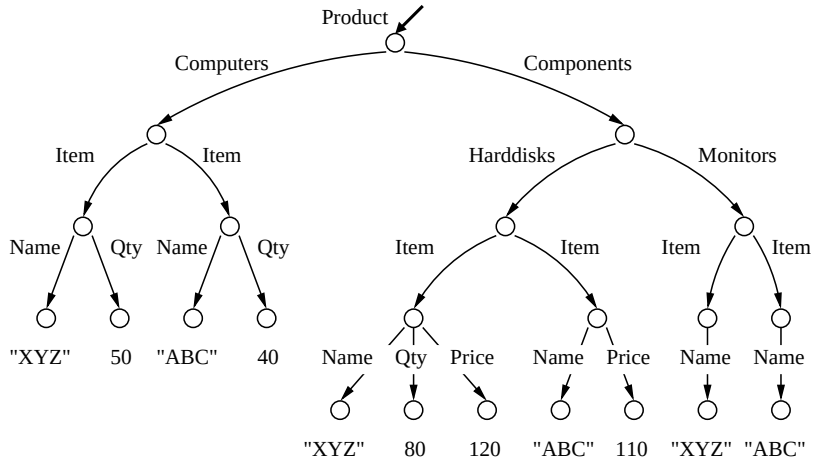


Figure 2: An Example XML Document Represent in Graph Model

- Retrieve all harddisk products.
  `/Product/Components/Harddisks/Item`
- Retrieve all products manufactured by *XYZ*.
  `/Product//Item[Brand = "XYZ"]`
- Retrieve all harddisks manufactured by *ABC*.
  `/Product/Components/Harddisks/Item[Brand = "ABC"]`
- Retrieve all components.
  `/Product/Components/*/Item`

3 Related Work

By using XQL [12] as a profile language, an efficient filtering mechanism that takes structure information into account for matching each subscription against XML stream data was presented in [3] using XPath [11]. However, the merging of similar subscriptions for further optimization was not addressed. Our work also has similarities with the very recent work proposed by [4, 10], in which the containment of XPath queries was investigated in detail. In particular, a new data structure based on the string Trie index was proposed in [4]. Their proposed data structure is similar to ours in that paths are encoded in a directed acyclic graph. However, ours differs in the handling of wildcards and descendant operators. [10] focused mainly on the tractability and analysis of methods for determining the containment of tree-pattern queries, in which XPath was selected as the query language. It described how to determine the containment of XPath queries efficiently, but did not explore the merging and handling of contained queries. Furthermore, both works did not address the problem from a mobile synchronization perspective. Hence, containment of queries was not applied for clustering clients into groups according to their subscription interests such that their updates are efficiently synchronized within each group. More importantly, selective propagation of updates (e.g., based on the containment of updates and subscriptions of different groups of clients) was not addressed.

Furthermore, our work also shares similar motivations with several other efforts including [1,2,5,9, 15, 16]. However, all these efforts only considered primitive or less expressive subscription languages. For instance, [2] considered conjunctions of simple event predicates, where each event is checked against an attribute value. Although efficient index structures for selective dissemination was presented

in [15], only the boolean model was considered. In [5], efficient algorithms for merging geographic queries were proposed. However, an efficient data structure for handling merged subscriptions was not addressed.

Similar motivations can also be found in mobile database applications. In [9], scalability is enhanced by grouping mobile clients according to their interests in sharing data in relational databases. A similar concept was recently applied for efficiently maintaining replica in an intermittently connected environment in [16]. With the exception of [5], the works above did not attempt to reduce costs by automatically merging similar queries. Finally, an extensive survey on recent research and development related to semistructured and web data, ranging from data models to query languages to database systems, was presented in [7]. Information regarding recent standards, techniques, and systems can be found at many XML portals such as `xml.com` and `xml.org`.

Other noteworthy mobile computing work, include Bayou [6] and Deno [8], which focus on conflict resolution and consistency maintenance. These works use mechanisms such as compensating transactions and voting protocols to enforce constraints. Moreover, as the number of clients maintained by each server increases, clients must be serviced in groups in order to maintain scalability. Broadcast databases [1] addressed this problem in the wireless domain but is primarily aimed at reducing the response time for data requests.

4 Overview of Solution

This section provides a basic idea of how the proposed query merging mechanism works. The key to our solution is an efficient mechanism to determine if two XQL expressions are overlapping. Overlapping expressions are merged so that the server can process fewer updates and the amount of information sent may be reduced (e.g., by exploiting the advantages of multicasting). However, we assume here that the client applies a post-filtering query over the received data in order to perform the update to its local data.

Two XQL expressions are considered overlapping if the XML segments retrieved from the same XML document by these two expressions are also overlapping or completely contained from one to another.

Consider the following example of computer hardware sales force automation system:

- `/Product/Computers` $\supset$
  `/Product/Computers/Item[Brand = "XYZ"]`

A computer item with brand name *XYZ*, represented by a path element `Item`, is a child element of element `Computers`. Their relationship is reflected in the above XQL expressions. In this case information regarding that `Item` should be delivered to both subscribers. However, the first subscriber is interested in more general computer product, which may or may not be interesting to the second subscriber depending on whether the expression is about `Item`. Consider another example below:

- `/Product/Computers` $\cap$ `/Product/Components` $= \phi$

These two subscriptions are mutually exclusive (since elements `Computers` and `Components` are two distinct children of element `Product`) so they cannot be merged. Similarly, even though both subscriptions below are interested in the `Item` elements, they cannot be merged as these two `Item` elements are under two independent (i.e., mutually exclusive) parents (`Computers` and `Components`).

- `/Product/Computers/Item` $\cap$
  `/Product/Components/Harddisks/Item` $= \phi$

wildcard (`*`) is more general than any literal tokens since it can match any literals within the specified scope. For instance, the wildcard below can match any child elements of `/Product`, including the element `Components`.

- `/Product/*` $\supset$ `/Product/Computers`

When descendant operators are involved, we cannot determine whether two subscriptions are independent by observing the XQL expressions. However, we can still determine their dependency if DTD information is available. For instance, we can confirm the following two subscriptions are not independent as `Computers` contains `Brand`.

- `//Brand` $\cap$ `/Product/Computers` $\neq \phi$

Finally, further dependency information can be obtained by observing the predicates inside the XQL filter conditions. For example, the following two subscriptions are independent (because of their exclusive price ranges) although they are both interested in the child elements under `Product`.

- `//Product//Item[Price < 10]` $\cap$
  `//Product//Item[Price > 30]` $= \phi$

All XQL *method invocation*s like the `!count()` function are based on the full result set of element nodes, and all their disjoint relations cannot be determine statistically as the following example illustrates. Therefore *method invocation*s can be treated as though they do not exist.

- `//Item/Price!max()` $\cap$ `//Item/Price!min()` $= \phi$
- `//Item/Price!max()` $\cap$ `//Item/Price!min()` $\neq \phi$
  `(//Item/Price!count() = 1)`

4.1 Transactions from Other Computers

The update statement in SQL plays a crucial role to make the manipulation and transactions of data stored in relational databases convenient and expressive. While the original XQL proposal did not include any update capabilities, the extended XQL [14] supports a complete set of update constructs from create to copy and move. These constructs are implemented as functions in XQL and can be invoked as other standard XQL functions like ancestor() or count() using the '!' notation .

4.1.1 Constructors

New elements, attributes, or texts can be created interactively by the insert function. The function accepts a plain path (i.e. a path without filters or subqueries) as its only argument. The quoted string in the path will be treated as the value of a text or an attribute value. For example, the following will create an empty element Name under every Restaurant element:

```
*/Restaurant!insert(Name)
```

The following will create an Entree element under every Restaurant, then create a Name element under Entree, and finally create the text "Black bean soup" under Name:

```
*/Restaurant!insert(Entree/Name/"Black
                    bean soup")
```

Finally the example below will create an attribute Note with value "Sunday Only" for the second Entree of each Restaurant:

```
*/Restaurant/Entree[1]!insert(@Note/
                         "Sunday Only")
```

Similarly, there are insertBefore(path) and insertAfter(path) constructors to insert path as a sibling before and after the current reference node, respectively. For example, the following will insert an element Cafe on the same level as Restaurant, just before the second Restaurant:

```
Restaurant[1]!insertBefore(Cafe)
```

4.1.2 Delete

Delete can be executed by invoking the function delete() with no arguments. It will delete all the nodes (and their descendants) from the current context. For example, the following will delete all the Names (and their descendants) from Restaurant elements. Note that the Restaurant elements will not be deleted in this case.

```
Restaurant/Name!delete()
```

Another example below will delete everything from the current context. If the root context is set to the global entry point to the whole database, it will delete all data in the database.

```
*!delete()
```

4.1.3 Copy

Cloning elements, attributes, or texts is possible by using the copy function. It accepts one argument which is the source of the copying. It will copy all the nodes (and their descendants) from the resultant context set of the evaluated argument, to every node in the current context. For example, the following will copy the content of the first Entree of Restaurants in the whole repository to the second Entree of the Restaurants.

```
(Restaurant/Entree)[1]!copy(//(Restaurant
          /Entree)[0]/*)
```

Note that very node in the current context will get a copy of the argument path, so the following example will make a copy of the content of the first Entree to every Entree including the first Entree itself. The argument path will be evaluated every time against every node in the current context. We define the *sub path* of an operation to be the argument path of the operation.

```
Restaurant/Entree!copy(//(Restaurant/
                      Entree)[0]/*)
```

As with Create, there are copyBefore and copyAfter operations besides Copy, which will copy the source to be just before or after the reference node, at the same level.

4.1.4 Move

The move operation will move the resultant reference nodes from the evaluated argument path to become the child of the nodes in the current context. As with Create and Copy, there are moveBefore and moveAfter operations. For example, the following will move the Ratings of Entrees to just after the Price elements of Entrees:

```
*/Restaurant/Entree/Price!moveAfter(
           //Restaurant/Entree/Rating)
```

Note that before the actual move operation is executed, validity checking needs to be done to ensure that ancestor nodes are not being moved to become the descendant nodes of the nodes in the current context. Otherwise, nodes in the current context would become invalid and no longer accessible from the root entry point(s).

4.1.5 Update

The value of an element or attribute can be updated by using the update function with the new value as an input argument. For example, the following will update Name and Price of the 2nd Entree of each Restaurant to ”Onion soup” and ”2.04” respectively.

```
*/Restaurant/Entree[1]/
    Name/*!update("Onion soup")
*/Restaurant/Entree[1]/Price/*!update("2.04")
```

Update can also be used to update the tagname of an element. For example, the following will rename all the Restaurant tagnames to Cafes:

```
*/Restaurant!update("Cafe")
```

5 Data Structure and Algorithms

5.1 Merging Simple Path Expressions

A naive approach to merging subscriptions has very poor performance. For instance, whenever a new subscription is created, it needs to be checked against all existing subscriptions or groups of subscriptions to determine if it is overlapping with any of them.

To address this problem, we present a sophisticated index structure. This index structure is briefly illustrated by the diagram shown in Figure 3b.

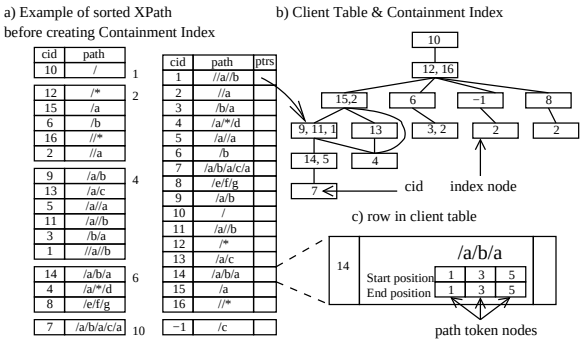


Figure 3: Data structure of Containment Index

With this index structure, we are able to improve the performance of merging subscriptions substantially as it captures the subscription containment relationships between clients. The Containment Index is a directed acyclic graph (in practice, it is a tree with some *index nodes* pointed to by more than one parent nodes). Each *index node* holds a list of client identifiers (*cid*). Each cid uniquely identifies a subscription client. The parent-child relationship of the index structure represents the subscription containment relationship, in which the data interested by each *cid* of an *index node* is a superset of the data interested by the *cid*s of all its child *index nodes*. In other words, data of interest to the *cid*s of an *index node* are also of interests for those *cid*s of its parent *index node*. *Cid*s held by the same *index node* implies equivalence, i.e., the clients share the same interest or subscription.

Each *index node* contains the following variables:

Cids: Client Identifiers. Note that the maximum number of client identifiers an *index node* can hold depends on an adjustable, predefined constant.

Next Pointer: For performance and efficient implementation of the paging mechanism, *index node* is implemented as a fixed size block. If the number of *cid*s exceeds the maximum number allowed, another *index block* will be created and it will be chained to the current block using *next pointer* (in a linear manner).

Running level: Every XQL expression of *cid*s in the same *index node* has the same number of *path token*s. Running level is an integer value to represent the number of tokens each XQL expression has, for each *cid* in the same index node. The *running level* of an XQL expression containing descendant operators is treated as if the expression was expanded with respect to the schema of the document (e.g., DTD).

Note that the *running level* of an *index node*, which has XQL expressions containing descendant operators, changes at run-time depending on the other path expressions in the Index. This is based on the assumption that no schema is provided.

It is a requirement of the Containment Index that each *path token*, for a given path, has to be represented by an index node. For example, the Containment Index in Figure 3b has an *index node* which contains *cid* = -1. In this case, we suppose that `/c/a` exists in the database, hence, the index node containing *cid* = -1 acts as a 'dummy' node for the path token `/c` for *cid* = 2.

5.1.1 Tokenization

The XQL parser used in our prototype development is an *event-based* parser which breaks XQL expressions into *path tokens* via callback functions. Each wildcard operator ($*$), child operator ($/$), descendant operator ($//$) and literal (e.g. `Stock`) is considered as a single *path token*. Although predicates ($[\,]$) need to be checked for subscription dependency, they will be treated separately using the technique similar to the one presented in [2].

The index structure also stores the parse tree of each XQL expression. Common subscriptions can be located in constant time using the *Client Table* (Figure 3b). The *Client Table* stores basic information about each client as well as its XQL expressions.

A *path token* node contains the following variables (see Figure 3c) along with other runtime variables based on the parse tree structure:

Start Position: The starting character's position of the representing *path token* in the XQL literal string.

End Position: The last character's position of the representing *path token* in the XQL literal string. Together with the Starting Position attribute, The end position can be utilized for string comparisons.

Filters and Predicates: There is a list of parsed predicates from the XQL parse tree. This list allows the index engine to further refine the disjoint detection mechanism, especially for those with predicates.

Each tokenized XQL expression is annotated with its total number of tokens, which is the *running level* of the expression. To construct the index from a set of existing subscriptions, all XQL expressions will first pass through the tokenizer. They are then sorted by the number of *path tokens* in increasing order. The second criteria for the sorting is by the following *path token* order:

$$\text{path op} \rightarrow \text{descendant op} \rightarrow \text{wildcard op} \rightarrow \text{literal}$$

Figure 3a shows a sorted list of tokenized XQL expressions that respects the above ordering. The sorting will enable construction of the whole Containment Index in effective order.

5.1.2 Insertion

When a tokenized XQL expression is inserted into the Containment Index, it starts from the root *index node* and keeps track of the current *running level* (`l`) variable. This is necessary as the Contain-

ment Index represents the overlap between subscriptions, not the XQL expression itself. The depth of the Containment Index does not directly correlate to the position of *path token*s in the XQL expression that we are comparing. Therefore the *running level* is necessary to identify the token in the XQL token list that is being compared.

Traversing the Containment Index and inserting an XQL expression without wildcard and descendant operators is simple. We first describe the algorithm of insertion by assuming no wildcard or descendant operators are available. The algorithm can then be extended to include the handling of wildcard and descendant operators. For clarity, present the algorithms below using recursion, while their actual implementations use an iterative approach.

```
// return a new empty index node
INDEX-NODE-CREATE(cid)
 1 n ← ALLOCATE-INDEX-NODE();
 2 for i ← 1 to CIDMAX do
 3   n.cid[i] ← ∞;
 4 n.{parent, child, sibling, next} ← φ;
 5 n.cid[0] ← cid;
 6 n.runLvl ← client[cid].PE.size() - 1;
 7 return n;

CLIENT-INSERT(cid)
 1 T.root ← CONTAIN-INSERT(cid, T.root, 0);

// node is the root of subtree for insertion
// l is running level denotes which token to check
// Assume all inserting PEs are pre-sorted
CONTAIN-INSERT(cid, node, l)
 1 if node = φ then
 2   return INDEX-NODE-CREATE(cid);
 3 if node.runLvl = l then
 4     if IS-EQUIV-IN(cid, node, l) then
 5       if client[cid].PE.size() - 1 = l then
 6         node.insertCid(cid);
 7         client[cid].ptr ← node;
 8       else
 9         n ← CONTAIN-INSERT(cid, node.lastChild(), l+1);
10         if n ≠ node.lastChild() then
11           node.insertChild(n);
12       return node;
13     else
14       return INDEX-NODE-CREATE(cid);
15 else
16   if IS-EQUIV-IN(cid, node, l) then
17     n ← INDEX-NODE-CREATE(cid);
18     node.runLvl++;
19     n.insertChild(node);
20     CONTAIN-INSERT(cid, node, l + 1);
21     return n;
22   else
23     return INDEX-NODE-CREATE(cid);

IS-EQUIV-IN(cid, node, l)
1  while node ≠ φ do
2    for i ← 0 to CIDMAX do
3      if node.cid[i] = ∞ then
4        return false;
5      if client[node.cid[i]].PE.token(l) ≡
6          client[cid].PE.token(l) then
```

```
7          return true;
8       node ← node.next;
8    return false;
```

When a *cid* is inserted to an *index node*, the reverse pointer in the *Client Table* for the current *cid* is also inserted for quick look up. Also when a new *index node* is created, its *running level* is set according to the *running level* of the given XQL expression.

5.2 Handling Wildcard/Descendant Operators

The pseudo-code above greatly simplifies the insertion of a client subscription to illustrate the main structure of the algorithm. This was done by disregarding all issues involving wildcard ($*$) and descendant ($//$) operators. A wildcard operator is treated as the parent for all literal operators if all their ancestors, without predicates, are equal. For example, `/a/b/*` is the parent of `/a/b/c`. During insertion, if the current path token of the XQL expression being inserted is a wildcard operator, instead of checking the node's last child, we need to perform the insertion on every child node. This idea is illustrated in the pseudo-code below.

```
     // insert after line 3 in CONTAIN-INSERT
 1   if client[node.cid[0]].PE.token(l) = '*' then
 2     if client[cid].PE.token(l) = '*' then
 3       if client[cid].PE.size() - 1 = l then
 4         node.insertCid(cid);
 5         client[cid].ptr ← node;
 6       else
 7         n ← c ∈ node.childs() s.t.  c.cid[0] = '*';
 8         if n ≠ φ then
 9           CONTAIN-INSERT(cid,n,l+1);
10         foreach c ∈ node.childs() - n do
11           CONTAIN-INSERT(cid,c,l);
12     else
13       n ← CONTAIN-INSERT(cid,node.lastChild(),l);
14       if n ≠ node.lastChild() then
15         node.insertChild(n);
16     return node;
17   else
```

Every XQL expression that contains descendant operators has to be checked against the schema of the XML documents in the server. This checking process involves the retrieval of all possible paths in the schema and inserting them accordingly. However as the elements of the schema form an acyclic graph and due to the nature of the inclusion, only the first occurrence of the expression in such cycles will be considered. For example, if `/a/b/a` exists in the schema, `//a` will expand as `/a` only.

5.3 Synchronization Engine

When a mobile client issues an update request and sends it to the XSync server, the Integration Module communicates with the XML Database. If the transaction is successful, it passes the mobile client identifier (*cid*) and the query (*q*) to the Synchronization Engine. The Synchronization Engine locates the pointers associated with *cid* in the Client Table. It then uses the pointer (or pointers if the XQL expression contains wildcard and/or descendant operators) to locate the *index node*s within the Containment Index which contains the client identifier. Notices that only XQL expressions with

wildcards and/or descendant operators will contain a list of pointers, other XQL expressions will only be pointing to a single node.

In the non-enhanced version of XSync, once the *index node* is found, all the client identifiers which are in *index nodes* that are ancestors and descendants of the original *index node* will be broadcast the update. Although this approach achieves a relatively good result, it can be greatly improved.

```
CLIENT-SEARCH(cid, q)
 1 C ← φ;
 2 for each ptr ∈ client[cid].ptr do
 3 node ← *ptr;
 4   // include equivalent PEs
 5   C ← C ∪ ∀c_i, c_i ∈ node.cid;
 6   for each c ∈ node.parents() do
 7     C ← C ∪ CLIENT-SEARCH-UP(c);
 8   if processLoad() > bandwidthLoad() then
 9     for each c ∈ node.childs() do
10       C ← C ∪ CLIENT-SEARCH-DOWN-ALL(c);
11   else
12     for each c ∈ node.childs() do
13       C ← C ∪ CLIENT-SEARCH-DOWN(c, q, 0);
14 return C;

CLIENT-SEARCH-UP(node)
 1 C ← node.cid;
 2 for each n ∈ node.parents() do
 3   C ← C ∪ CLIENT-SEARCH-UP(n);
 4 return C;

CLIENT-SEARCH-DOWN-ALL(node)
 1 C ← node.cid;
 2 for each n ∈ node.childs() do
 3   C ← C ∪ CLIENT-SEARCH-DOWN(n);
 4 return C;

CLIENT-SEARCH-DOWN(node, q, l)
 1 C ← node.cid;
 2 c ← node.cid[0];
 3 if client[c].PE.token(l) ≡ q.token(l) then
 4   for each c ∈ node do
 5     C ← C ∪ c;
 6   for each c ∈ node.childs() do
 7     C ← C ∪ CLIENT-SEARCH-DOWN(c, q, l+1);
 4 return C;
```

The equivalence binary operator (≡) always evaluates to true when comparing a wildcard operator to a literal string.

6 Enhancements to the Synchronization Algorithms

As all ancestor *index nodes* represent subscriptions to data that are supersets of the subscription *index node* iteself (without considering predicates), it is necessary to forward all updates performed by a client to it's ancestors. However, this is not the case for descendants. If the client subscription covers a large portion of the XML document, forwarding updates to all descendants will result in a large amount of communications between clients and the Synchronization Engine. However by combining a

mobile client's update query with its own subscription XQL expression, the Engine is able to compute a disjoint set in its decendant *index node*s.

For example, in Figure 3b, if client 15 issues an update operation:

```
Q15:  d/e/f!update("g")
```

The Engine can merge the update query with its XQL subscription to form a new path expression `/a/d/e/f`. It can then create the token nodes and match these against the descendants of the *index node* containing client 15. In this example, all descendants of *index node* will match as disjoint and thus all *cid*s in the subtrees are not considered as part of the set of broadcasting clients.

Using the update statement to match against the descendants of an *index node* is very similiar to the insertion of a query to the *Containment Index*.

For the purposes of the Synchronization Engine, the update statements detailed in Section 4.1 can be classified into two categories:

1. **Statements that do not affect other disjoint paths:** These statements include `!insert(PE)` and `!delete(PE)`. If the statements do not contain wildcards or descendant operators, the Engine executes `CLIENT-SEARCH-DOWN`. Otherwise, it expands the descendant operator to determine all unique paths from the DTD. For each of these paths the Engine searches the descendant *index node*s and checks the path tokens against the update statement *path token*s. Eventually either the *index node* reaches a leaf node or the update statement runs out of *path token*s. At that point, the current *index node* and its descendants are treated as affected.

2. **Statements that affect other disjoint sets:** In this situation, the Engine has to perform two separate steps of overlapping expression detection, hence increasing the runtime cost. Firstly, we need to check the overlap for the target path as mentioned above, then the sub path expression has to be treated as a separate update, searching from the root *index node* as with a normal search. All *cid*s located by the two overlapping expression detection mechanisms represent clients that are affected and have to be notified of the update. Examples of update statements in this category include `!move(PE)`, where `PE` is the sub path expression.

6.1 Update Merging

When mobile clients perform updates on their local cache of the database, they forward each update to the server so as to allow the server to forward the updates to appropriate clients. The server determines whether an update should be forwarded to a given client based on that client and the updating client's subscription. The server only forwards updates to those clients which are interested in the update.

Consider the situation where Client A has an overlapping subscription with Client B and Client C has an overlapping subscription with Client B. When both Client A and C perform updates to their local cache, their update operations are forwarded to the server. A naive solution to keeping Client B up-to-date would involve broadcasting two separate update operations to Client B. However, XSync performs a merge between the two operations and encapsulates the operations into a single message to Client B. Hence, this reduces the communication costs between the clients and server.

However, being able to merge several update operations from different clients into a single message leads to issues of conflict detection and resolution. In the situation where clients perform updates on the same subset of nodes remotely, their update operations may be conflicting in terms of their target and/or sub path. Hence, conflict detection is necessary to merge the updates of several mobile clients.

To analyse the problem of update merging, we first consider a specific example involving two clients. We next generalise our analysis to merging the update operations of *n* clients that are forwarded to the server.

Consider the simplified problem where two clients (Client A and Client B) have an overlapping subscription, where both have *cid*s in the same *index node*, and each perform an update operation on their local database. The updates performed by Client A and Client B are forwarded to the server and it is the responsibility of the server to detect and resolve any conflicts, while forwarding these updates to the appropriate clients.

To do this, the server constructs a Containment Index structure similar to Figure 3b, using the algorithm described in Section 5.1 and 5.2. In contrast to Figure 3b, the Containment Index captures the containment relationships among the update operations performed by the clients. Hence, instead of *cids* stored in each *index node*, *oids* are stored. The server also maintains an *Operation Table* (similar to the *Client Table*) which contains basic information about each operation including:

Oid: The Operation Identifier of the update operation. Each operation has a unique *Oid*, hence, an operation with a sub path has a different *Oid* from the operation with its target path.

Cid: The Client Identifier of the client which performed the operation. This allows the identification of conflicting operations between clients.

Operation: The type of operation that was performed on the target path (e.g. `!insert(PE)`).

Target Path: A value which indicates if the path being represented is the target path or the sub path. This aids in the conflict detection of overlapping paths.

6.1.1 Types of Conflicts

Given two edit operations, a Conflict occurs if and only if they have paths that are overlapping. We define two *disjoint* subclasses of Conflict:

Direct Conflict (DC): A DC is a Conflict such that the order that the operations are carried out is important. That is, if update operations `x` and `y` are in DC then one of the operations, `x` or `y`, cannot be performed if the other operation is performed first. For example, let `x` be an insert operation and `y` be a delete operation. If `y` is performed first, `x` cannot be performed as it deals with a node that has already been deleted.

This situation occurs when one of the operations in DC is `update()` or `delete()` and their target paths are in Conflict, or one of the operations is a move operation and its subpath is in conflict with the path of the other operation. Note that the `update()` operation may participate in a DC, as the operation can update the tagname of an element.

Syntax Conflict (SC): A SC occurs when two update operations are Conflicting in terms of their target path or (if applicable) the sub path of one of the operations is Conflicting with a path of the other operation. The order in which two SC operations are performed on the database affects the resulting database as we are dealing with the ordered model.

Table 2 lists the update operations that are in SC, given that the operations are in the same *index node*. *op* indicates an update operation including *insert*, *move* and *copy*.

	op	*op*After	*op*Before
op	Yes	No	No
*op*After	No	Yes	Yes
*op*Before	No	Yes	Yes

Table 2: SC between update operations in the same *index node*.

It is noteworthy that in Table 2, *op*Before is in SC with *op*After. This occurs in the situation where one of the operations is position specific. For example, let `x = a/b[0]!insertAfter(c)`

and `y = a/b[0]!insertBefore(b)`. `x` and `y` are in SC because if `x` is performed first, then `y`, the resulting database would be different from when `y` is performed before `x`.

Note that despite the ordering of a pair of update operations that are in SC, the operations can still be applied ot the database. This is not the case for operations in DC.

6.1.2 Conflict Detection & Resolution

Once the update operations of Client A and Client B have been processed to construct the Containment Index, we traverse the data structure to identify path conflicts. This is similar in concept to the steps carried out by the server in response to the update by client 15 at the beginning of Section 6. By constructing the Containment Index based on Client A and B's update operations, we are able to detect conflicts.

In order to resolve conflicts, we have to consider each subclass of Conflicts individually.

Direct Conflict (DC): The architecture of XSync implicitly orders the update operations that it receives from its clients. That is, the server receives the update operations serially. Hence, for operations that are in DC, if the operations are received in an order such that both operations can be performed to the database in that order, then the conflict has been resolved.

On the other hand, if the order in which the DC operations arrive at XSync result in one of the operations not being able to be performed on the database, XSync provides a resolution to this conflict. On detection of such a conflict, XSync selects an operation (out of the two in DC) to undo based on the DC resolution rules listed below:

1. A `delete` operation is always selected to be undone over any operation.
2. A `move` operation is always selected to be undone over any operation if Rule 1 does not apply.
3. A `update` operation is always selected to be undone over any operation if Rule 1 and 2 do not apply.

The rules are listed in order of precedence. That is, Rule 1 is evaluated first and if it does not apply, Rule 2 is evaluated, etc.

The intuition behind the resolution rules listed above is to undo the operation which has a more 'costly' effect on the database. For example, delete operations are always chosed by XSync to be undone because the operation, if executed, would result in a significant amount of data being removed from the database.

Syntax Conflict (SC): As XSync receives update operations from its clients in a serialized manner, the order that the SC operations are performed has already been resolved. However, given this serialized order, some operations that are in SC still may not be able to be applied directly onto the database.

For example, given two *insert* operations with the same target path, one of the operations will have to be modified syntactically to allow it to be performed on the database. This is because, after the first operation has been executed, the target path is no longer a leaf node and hence an insert operation cannot be performed on it (rather the operation has to be changed to an insertAfter with a modified target path).

After all the conflicts have been resolved, we forward the merged sequence of update operations to all clients that have overlapping subscriptions with the initiating client(s).

Note that the responsibility for conflict detection and resolution is with XSync. That is, clients need not provide facilities to deal with conflicts. Hence, this reduces the complexity of the clients that communicate with the server.

6.1.3 Multiple Clients

The above solution can be generalized to *n* number of clients. In this case, the Synchronization Engine maintains a single Containment Index for all client updates that arrive at the server. The Engine periodically broadcasts update operations to the appropriate clients based on the Containment Index for operations. Once the appropriate clients have been notified, the corresponding operations can be deleted from the Index.

The Containment Index is constructed by extracting all paths associated with each client's update operation, passing each path through the tokenizer (detailed in Section 5.1.1), sorting the tokens according to *path token* order and finally inserting each path into the Index.

The Engine first executes `DETECT-CONFLICTS` to detect and resolve any conflicts. It then executes `UPDATE-MERGE` to broadcast the update operations issued by the clients so far to all applicable clients.

Note that this solution also scales to the situation where a client issues several update operations to the server before the server broadcasts the updates to all the appropriate clients.

6.1.4 Algorithm for Detecting & Resolving Conflicts

The algorithm below detects conflicts between *n* client update operations.

```
// C is the set of clients that forwarded update
// operations to the server
DETECT-CONFLICTS(C)
1  dc ← φ
2  for each cid ∈ C do
3     cidSet ← CLIENT-SEARCH(cid) - {cid};
4     dc ← dc ∪
         DETECT-DIRECT-CONFLICT(T.root(), cid, cidSet);
5  RESOLVE-DIRECT-CONFLICT(SORT(dc));
6  for each cid ∈ C do
7     cidSet ← CLIENT-SEARCH(cid) - {cid};
8     DETECT-SYNTAX-CONFLICT(T.root(), cid, cidSet);

// L is a list that contains pairs of
// operations that are in DC. It is constructed
// such that the first of each pair is the
// operation that has to be undone later.
// Also, DCs with delete operations are
// considered first followed by move
// operations and finally update operations.
DETECT-DIRECT-CONFLICTS(node, cid, cidSet)
 1 C ← φ;
 2 N ← FILTER(cid, node.oid());
 3 {D, M, U} ← [];
 4 for each o ∈ N do
 5    if operation[o].op() = "delete" then
 6       D.append(o);
 7    elif operation[o].op() = "move" ∧
 8       !operation[o].targetPath then
 9       M.append(o);
10    elif operation[o].op() = "update" then
```

```
11      U.append(o);
12  L ← D;
13  L.append(M);
14  L.append(U);
15  for each n ∈ node.childs() do
16    for each o ∈ FILTER(cidSet, n.oid()) do
17      for each l ∈ L do
18        if operation[o].order > operation[l].order ∧
19          (l,o) ∉ C then
20          C ← C ∪ (o,l);
21    C ← C ∪ DETECT-DIRECT-CONFLICTS(n, cid, cidSet);
22  return C;

// S is a list obtained by ordering the
// output of DETECT-DIRECT-CONFLICTS
// according to cid of the operation that
// has to be undone due to DC and within that,
// by the order they were received
RESOLVE-DIRECT-CONFLICTS(S)
 1  undoCids ← [];
 2  for each (undoOp, conflictOp) ∈ S do
 3    if ∃(cid, order) ∈ undoCids s.t.
 4      operation[undoOp].cid = cid then
 5      continue;
 6    if ¬∃(cid,order) ∈ undoCids s.t.
 7      operation[conflictOp].cid = cid ∧
 8      operation[conflictOp].order > order then
 9      op ← operation[undoOp];
10      undoCids.append((op.cid, op.order));
11  for each (cid, o) ∈ undoCids do
12    for all op ∈ operation s.t.
13      op.cid = cid ∧ op.order ≥ o do
14      op.orig ← op.PE + op.op();
15      op.op ← "UNDO";

// Detect and resolve SCs.
DETECT-SYNTAX-CONFLICTS(node, cid, cidSet)
 1  N ← FILTER(cid, node.oid());
 2  L ← φ ;
 3  for each o ∈ N do
 4    if operation[o].op() != "update" ∧
 5      operation[o].op() != "delete" ∧
 6      operation[o].op() != "UNDO" then
 7      L ← L ∪ o;
 8  for each n ∈ node.childs() do
 9    for each o ∈ FILTER(cidSet, n.oid) do
10      for each l ∈ L do
11        if operation[o].op ≡_op operation[l].op then
12          if operation[o].order > operation[l].order then
13            OP-DELETE(o);
14            operation[o].changed ← true;
15            operation[o].orig ← operation[o].PE;
16            operation[o].PE ← NEW-PATH(l);
17            OP-INSERT(o);
18          else
19            OP-DELETE(l);
20            operation[l].changed ← true;
21            operation[l].orig ← operation[l].PE;
22            operation[l].PE ← NEW-PATH(o);
23            OP-INSERT(l);
```

```
24   DETECT-SYNTAX-CONFLICTS(n, cid, cidSet);
```

To detect DCs involving the delete operation involves traversing through the Containment Index. All *cid*s that are in nodes which are descendants of a delete operation node are in DC. Similarly for the move and update operation. We then have to undo some operations (if applicable) in order to resolve the conflict. In order to handle the situation when a client forwards multiple update statments and one of the operations have to be undone in RESOLVE-DIRECT-CONFLICTS, we keep track of the clients (*undoCids*) which have operations that have to be undone.

We define a function FILTER(*cid, Ops*), for use in DETECT-DIRECT-CONFLICTS, which returns a subset of *Ops* which were operations performed by *cid*.

We also define $\equiv_{op}$ to be a function in DETECT-SYNTAX-CONFLICTS that returns true if and only if its arguments correspond to a 'Yes' entry in Table 2.

OP-DELETE deletes the argument oid from the Containment Index, while OP-INSERT inserts the argument into the Containment Index. The implementation for OP-INSERT is equivalent to CLIENT-INSERT.

We also define extra parameters in the operation table for each operation to maintain information on whether the operation was modified in order to resolve a SC and if so, it's original target/sub path. We also include the *order* that the operation arrived at the server in order to resolve operations that are in conflict. NEW-PATH generates the new path that should be the target/sub path of *oid* in order to resolve the SC.

6.1.5 Algorithm for Update Merging

On receipt of *m* update operations from *n* clients, *C*, XSync performs the algorithm detailed above to detect and resolve conflicts. XSync then executes UPDATE-MERGE(L) where L is the list of operations returned by DETECT-CONFLICTS(). We assume that the client has facilities to undo specific operations.

We handle clients in *C* differently from clients with overlapping subscriptions with clients in *C*. This is because operation(s) have already been carried out on their local cache database, and hence may need to be undone to maintain consistency.

```
UPDATE-MERGE(L)
 1 message ← []
 2 for each o in L do
 3   if operation[o].op() = "UNDO" then
 4     op ← operation[o];
 5     message[op.cid] ← UNDO(op.orig) + message[c];
 6     continue;
 7   C ← CLIENT-SEARCH(o.cid, o);
 8   for each c in C do
 9     if c = o.cid then
10       if operation[o].changed then
11         op ← operation[o];
12         message[c] ← UNDO(op.orig) + message[c] + o;
13       else
14         message[c].append(o);
15     else
16       message[c].append(o);
17 for each c in message do
18   SEND-UPDATE(message[c], cid);
```

7 Performance

7.1 Settings

In this section, we present our cost model for calculating the worst and average case scenario. Let $|\mathrm{P}|$ denotes the total number of XQL expressions, D_{avg} as the average number of *path tokens* in all expressions calculated by $\frac{\sum_{p \in P} D_p}{|\mathrm{P}|}$ and S is the number of *cid*s stored in one *index node*.

7.1.1 XQL Expression Insertion Cost

Tokenization and Sorting of the original XQL expression set P is $O(|\mathrm{P}| \cdot log|\mathrm{P}| \cdot \mathrm{D}_{avg})$ and the insertion cost to the Containment Index is bounded by $O(\mathrm{S} \cdot \mathrm{D}_{avg} \cdot |\mathrm{P}|)$. Therefore, the cost of generating the whole index is:

$$O\Big(|\mathrm{P}| \cdot log|\mathrm{P}| \cdot \mathrm{D}_{avg} + \mathrm{S} \cdot \mathrm{D}_{avg} \cdot |\mathrm{P}|\Big)$$

7.1.2 XQL Expression deletion Cost

Trivial. Near constant time.

7.1.3 Search Cost

$$O\Big(\mathrm{C} \cdot \mathrm{S} \cdot fanout_{avg}()^{\mathrm{D}_{avg} - \mathrm{D}_{\mathrm{P}_{cid}}}\Big) + O\Big(\mathrm{S} \cdot fanin_{avg}()^{\mathrm{D}_{\mathrm{P}_{cid}}}\Big)$$

Practically, both $fanin_{avg}()$ and $fanout_{avg}()$ are very close to one.
C = 1 when the Synchronization Engine broadcasts to all descendants.

7.2 Experimental Results

The description of the experiment parameters are shown in Table 1.

Parameter	Range	Description
$\|Q\|$	1 - 1M	# of queries
q_{length}	0 - 10	Length of query
u_{length}	0 - 10	Length of update
$\|I\|$	1 - 20	Avg # of *cid*s / *index node*
$\|DTD\|$	1 - 100	Size of DTD
$Max(X_{\text{length}})$	10	Max. Depth of XML element

For simplicity, we assume each mobile client ($c_i \in C$) contains only one query ($q_i \in Q$) to describe its cache. q_{length} is defined as the total number of *path tokens* in a query and u_{length} is defined as the total number of *path tokens* in the update statement that a particular client issues. The size of the *DTD* is defined as the total number of unique element names in the Document Type Definition.

We develop a random DTD generator which generates DTD with $|DTD|$ as input to limit the number of element names. The DTD that is generated also allows for cyclic inclusion of element names for testing the correctness of descendant operator.

Also we made our own simple random XQL expression generator which can produce client queries and updates according to the q_{length} and u_{length} length parameters.

Queries that are generated using q_{length} are used to populate the index and they do not contain extended *method invocations* such as `!move(`PE`)` or `!delete(`PE`)`. Also the XQL expression

generator accepts extra parameters to limit the number of wildcard, descendant operators and their position within the expression.

The maximum depth of XML elements in the XML document has a direct relationship with the length of the query and the length of update. As any mobile client can specify its subscription to be the leaf node of the XML tree, q_{length} cannot exceed $Max(X_{\text{length}})$. Also, the combined length of the update and query of a particular client cannot exceed $Max(X_{\text{length}})$.

For example, in Figure 4 and Figure 5, the length of an update is fixed, therefore only clients with $q_{\text{legnth}} < Max(X_{\text{length}}) - u_{\text{length}}$ would be considered.

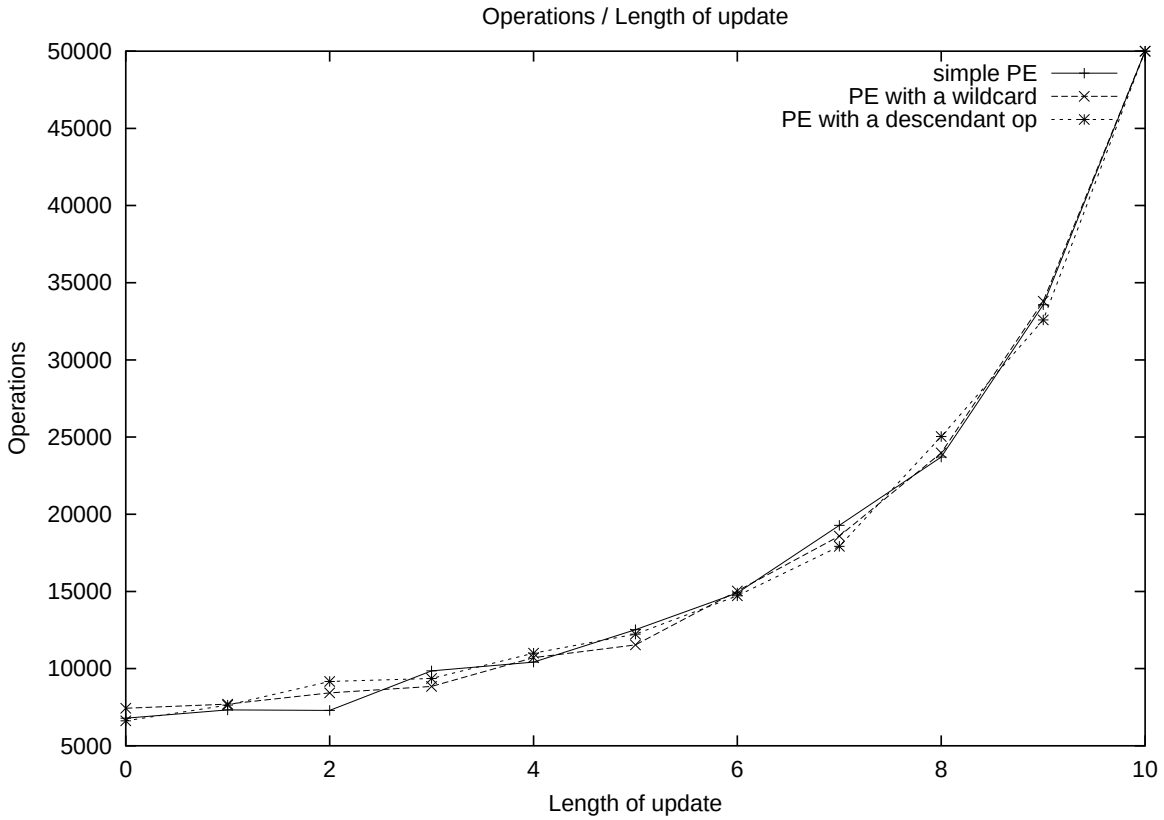


Figure 4: # of Operations / Length of Update

`CLIENT-SEARCH-DOWN-ALL`, $|Q| = 1M$,$|I| = 20$,$|DTD| = 100$,$Max(X_{\text{length}}) = 10$,$q_{\text{length}} = 0..Max(X_{\text{length}}) - u_{\text{length}}$

Figure 4 shows the number of operations required to search for overlapping clients when a client sends an update statement to the server. The number of queries are fixed at 1,000,000 and the method of search only utilises client subscriptions, ignoring client updates. In this situation, the server forwards updates to both ancestors and *all* its descendants.

Simple PE refers to XQL expressions which do not contain wildcards or descendant operatiors. We also specifically chose to test path expressions which included only one wildcard or descendant operator for both the client query and its update. However we did not limit the location of such an operator within the query or update. By doing so, we were able to analyse the behaviour of the most complex operators in the expression.

As the length of an update is fixed at u_{length}, we randomly chose a client with its q_{length} between 0 (e.g. /) and $Max(X_{\text{length}}) - u_{\text{length}}$. Therefore, this graph shows a scalarability of $O(u\text{length}^2)$. However, u_{length} can only approach $Max(X_{\text{length}})$ as any updates that are any longer will always yield

no result. Hence, the worst case is the same as simple broadcasting. This is applicable when the client's view is the whole document and the update statement is ignored.

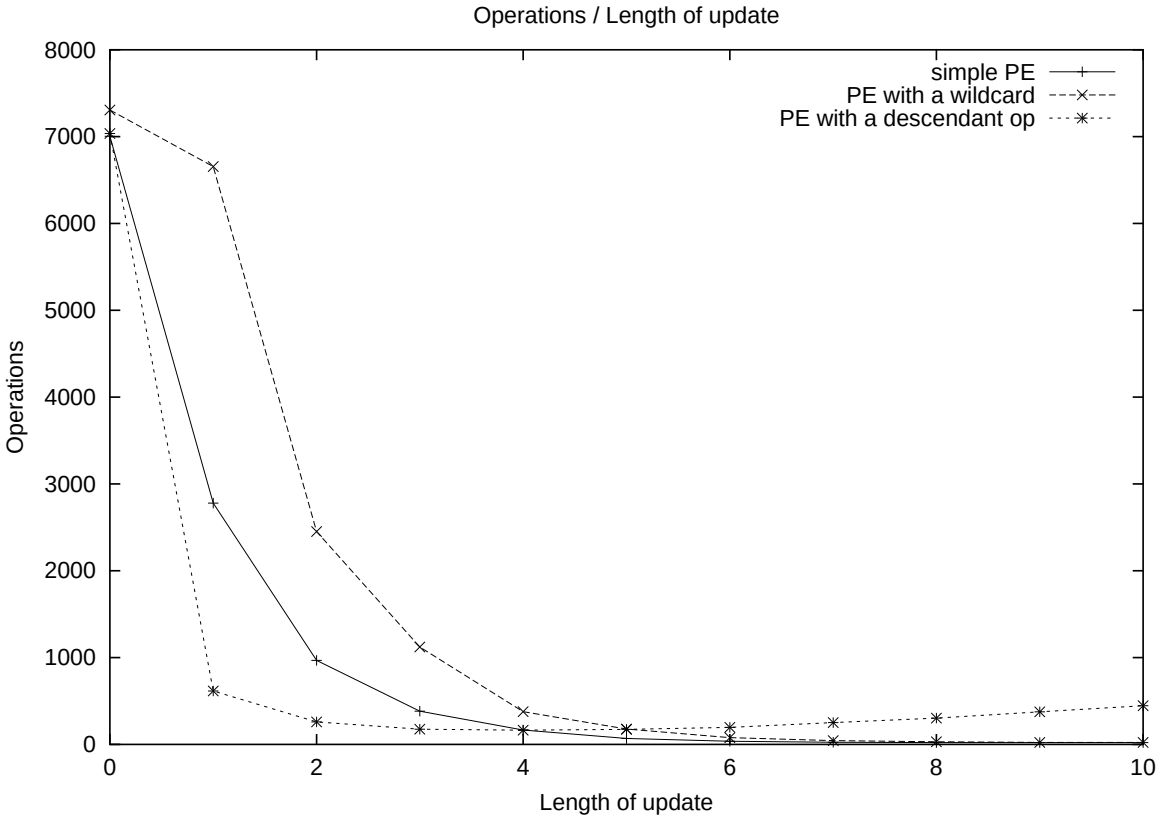


Figure 5: # of Operations / Length of Update

`CLIENT-SEARCH-DOWN`, $|Q| = 1M$,$|I| = 20$,$|DTD| = 100$,$Max(X_{\text{length}}) = 10$,$q_{\text{length}} = 0..Max(X_{\text{length}}) - u_{\text{length}}$

In contrast to Figure 4, Figure 5 shows the number of operations required to search for overlapping clients by matching the path expression to the update statments issued by the client. Comparing this with Figure 4, both have similar costs when u_{length} is equal. By checking the update statement, the cost is significantly reduced when the length of the update statement increases.

Instead of fixing the length of update statement and randomly choosing client queries, Figure 6 and Figure 7 shows the behaviour of the system when the length of the update statement is random. It is interesting to note that although the length of the query and length of the update has an inverse relationship in these graphs, Figure 5 and 7 illustrate a similar curve, unlike Figure 4 and 6. This is because when the length of the update or the length of the query is short, the overall length, on average, with the chosen counterpart is short as well. Together with the randomness for the location of the operator, it is more likely that a large subtree is included.

Figure 8 and 9 shows the growth of the search time against the number of queries. q_{length} and u_{length} is chosen at random and the average number of operations it taken. It is observed that the growth of the search time is logarithmic. In addition, the size of the DTD was chosen relative to the number of queries. This is the main factor resulting in the growth in the number of operations for complex queries (such as those which contain the descendant operator).

Figure 10 and 11 illustrates that as the size of the DTD increase, the number of expressions that can be obtained by expanding an update operation with descendant operators decreases, and hence reducing the cost of a search.

Notice that from Figure 8 to Figure 11, the performance of PE with a wildcard is only slightly

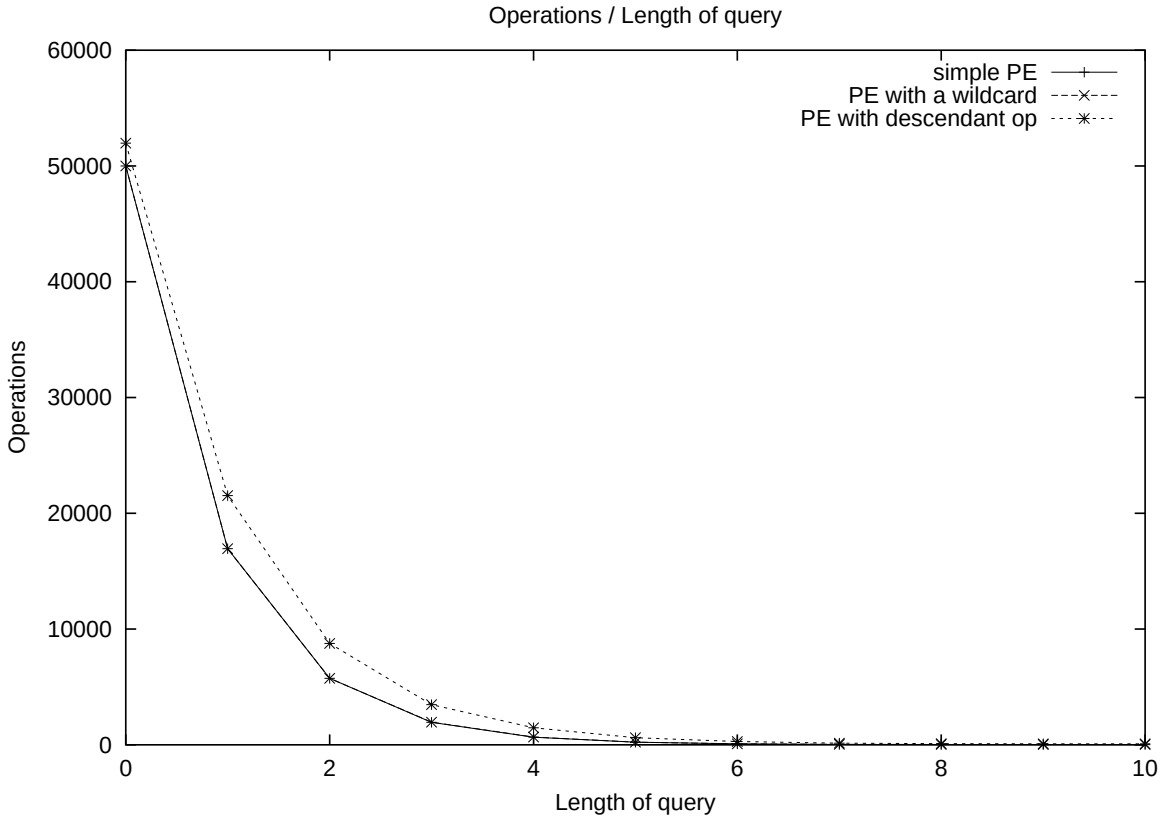


Figure 6: # of Operations / Length of Query

`CLIENT-SEARCH-DOWN-ALL`, $|Q| = 1M$,$|I| = 20$,$|DTD| = 100$,$Max(X_{\text{length}}) = 10$,$u_{\text{length}} = 0..Max(X_{\text{length}}) - q_{\text{length}}$

higher than the simple PE because q_{length} is choosen to be from 3 to 5. Other parameters were also set at such realistic values. However if we increase the fanout value by lowering the depth of the XML Document ($Max(X_{\text{length}})$) or by reducing the size of the DTD, the cost of searching wildcard queries and updates increases. Decreasing q_{legnth} will also have the same effect, as shown in Figure 7.

8 Conclusions

In this paper, we presented an efficient synchronization server for handling mobile XML data. The proposed server, XSync, consists of an Integration Module (for communication with the XML database) and a Synchronization Engine (for handling all synchronization issues). The Synchronization Engine utilizes a sophisticated index structure, which provids a significant improvement compared to current methods available. We also explored several enhanced synchronization algorithms for update merging and disjoint predicates and ranges, to further improve the performance of the system.

References

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: Data management for asymmetric communication environments. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1995.

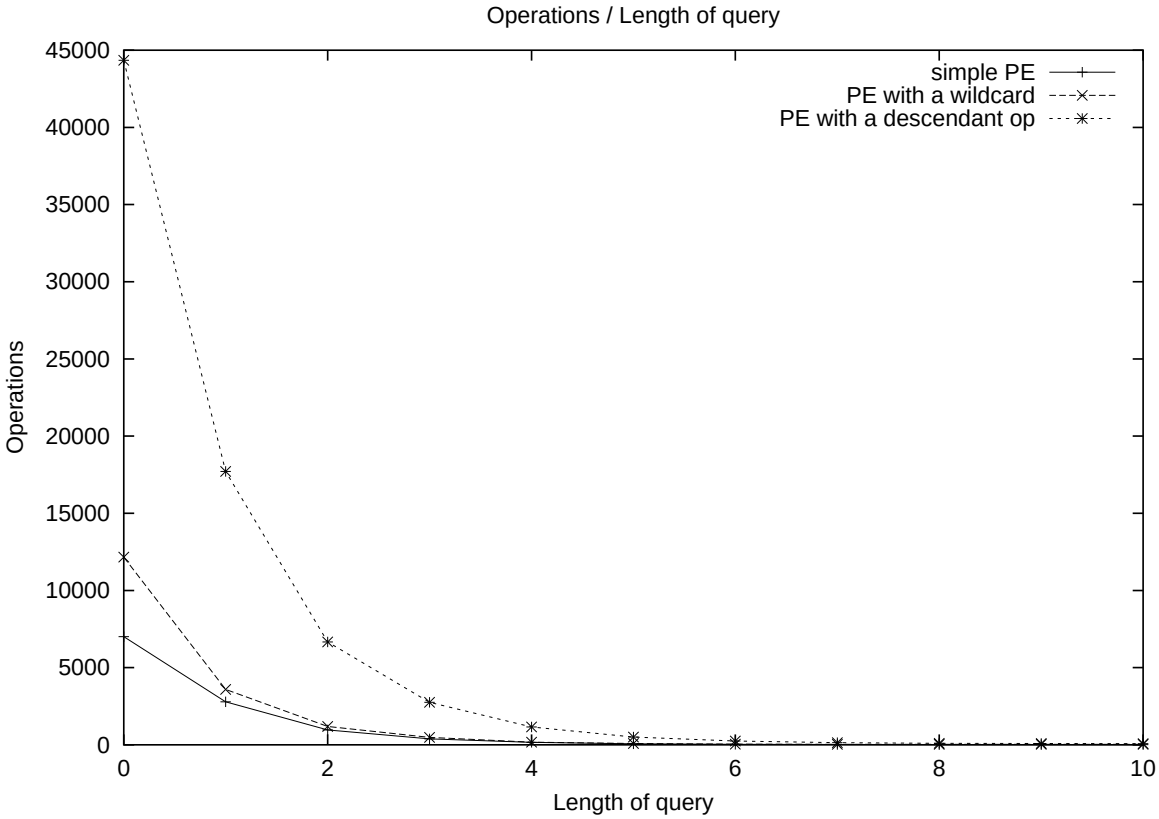


Figure 7: # of Operations / Length of Query

`CLIENT-SEARCH-DOWN`, $|Q| = 1M$,$|I| = 20$,$|DTD| = 100$,$Max(X_{\text{length}}) = 10$,$u_{\text{length}} = 0..Max(X_{\text{length}}) - q_{\text{length}}$

[2] M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra. Matching events in a content-based subscription system. In *Proceedings of ACM PODC*, pages 53–61, 1999.

[3] M. Altinel and M.J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In *Proceedings of the 26th VLDB Conference*, pages 53–64, 2000.

[4] C.Y. Chan, P. Felber, M.N. Garofalakis, and R. Rastogi. Efficient filtering of xml documents with xpath expressions. In *Proceedings of IEEE International Conference on Data Engineering*, February 2002.

[5] A. Crespo, O. Buyukkokten, and H. Garcia-Molina. Efficient query subscription processing in a multicast environment. Technical report, Stanford University, 1999.

[6] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The bayou architecture: Support for data sharing among mobile users. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, 1994.

[7] D. Florescu, A. Levy, and A. Mendelzon. Database techniques for the world-wide web: A survey. *SIGMOD Record*, 27(3):59–74, 1998.

[8] P.J. Keleher and U. Cetintemel. Consistency management in deno. *Journal on Special Topics in Mobile Networking and Applications (MONET)*, 1999.

[9] S. Mahajan, M.J. Donahoo, S.B. Navathe, M. Ammar, and S. Malik. Grouping techniques for update propagation in intermittently connected databases. In *Proceedings of the IEEE International Conference on Data Engineering*, February 1998.

[10] G. Miklau and D. Suciu. Containment and equivalence of xpath expressions. In *Proceedings of ACM Principles of Database Systems (PODS)*, 2002, to appear.

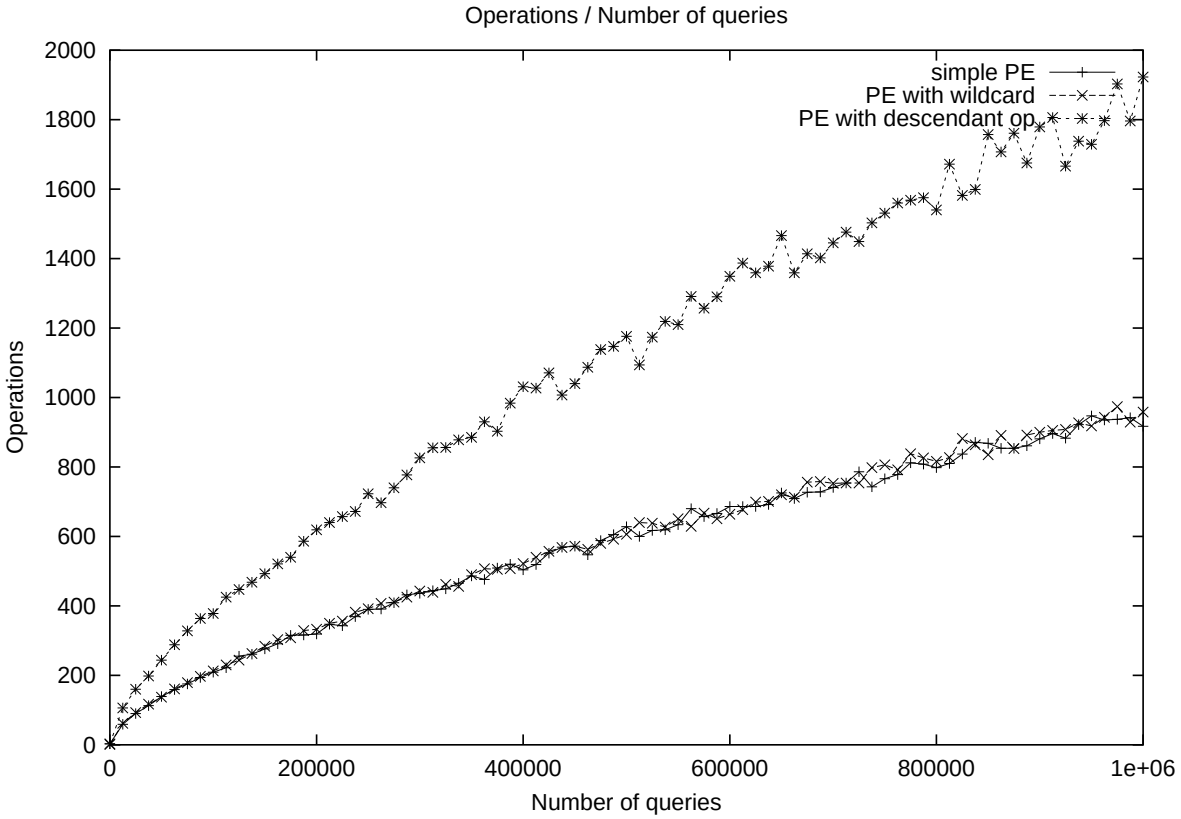


Figure 8: # of Operations / Number of Queries

`CLIENT-SEARCH-DOWN-ALL`, $|I| = 20$,$|DTD| = 10$,$Max(X_{\text{length}}) = 10$,$q_{\text{length}} = 3..5$,$u_{\text{length}} = 0..Max(X_{\text{length}}) - q_{\text{length}}$

[11] W3C Recommendation. Xml path language (xpath) version 1.0. *http://www.w3.org/TR/xpath*, November 1999.

[12] J. Robie, J. Lapp, and D. Schach. Xml query language (xql). In *The XSL Working Group, World Wide Web Consortium*, 1998. Available at `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.

[13] I. Tatarinov, Z.G. Ives, A.Y. Halevy, and D.S. Weld. Updating XML. In *SIGMOD Conference*, 2001.

[14] R.K. Wong. The extended xql for querying and updating large xml databases. In *Proceedings of ACM Symposium on Document Engineering (DocEng)*, November 2001.

[15] T.W. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the boolean model. *ACM TODS*, 19(2):332–364, June 1994.

[16] W.G. Yee, E. Omiecinski, M.J. Donahoo, and S.B. Navathe. Scaling replica maintenance in intermittently synchronized mobile databases. In *Proceedings of ACM CIKM*, pages 450–457, 2001.

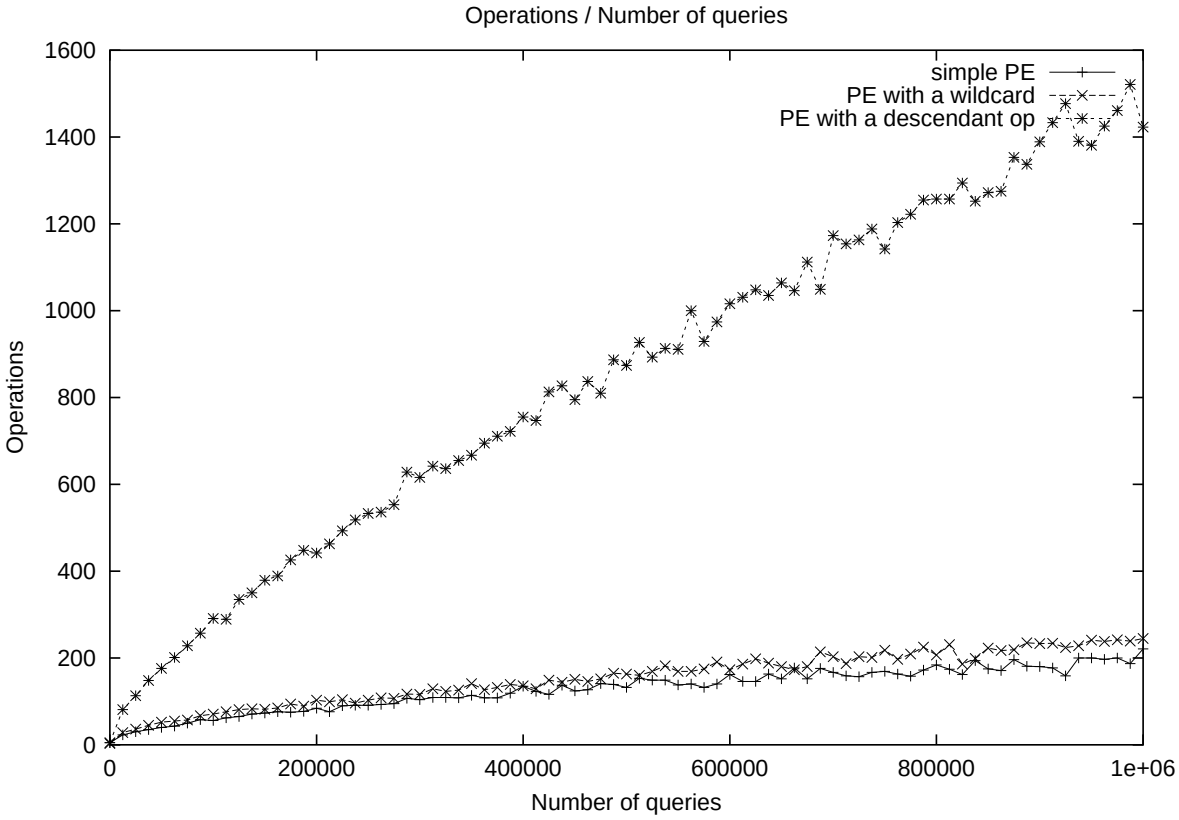


Figure 9: # of Operations / Number of Queries

CLIENT-SEARCH-DOWN, $|I| = 20$,$|DTD| = 10$,$Max(X_{\text{length}}) = 10$,$q_{\text{length}} = 3..5$,$u_{\text{length}} = 0..Max(X_{\text{length}}) - q_{\text{length}}$

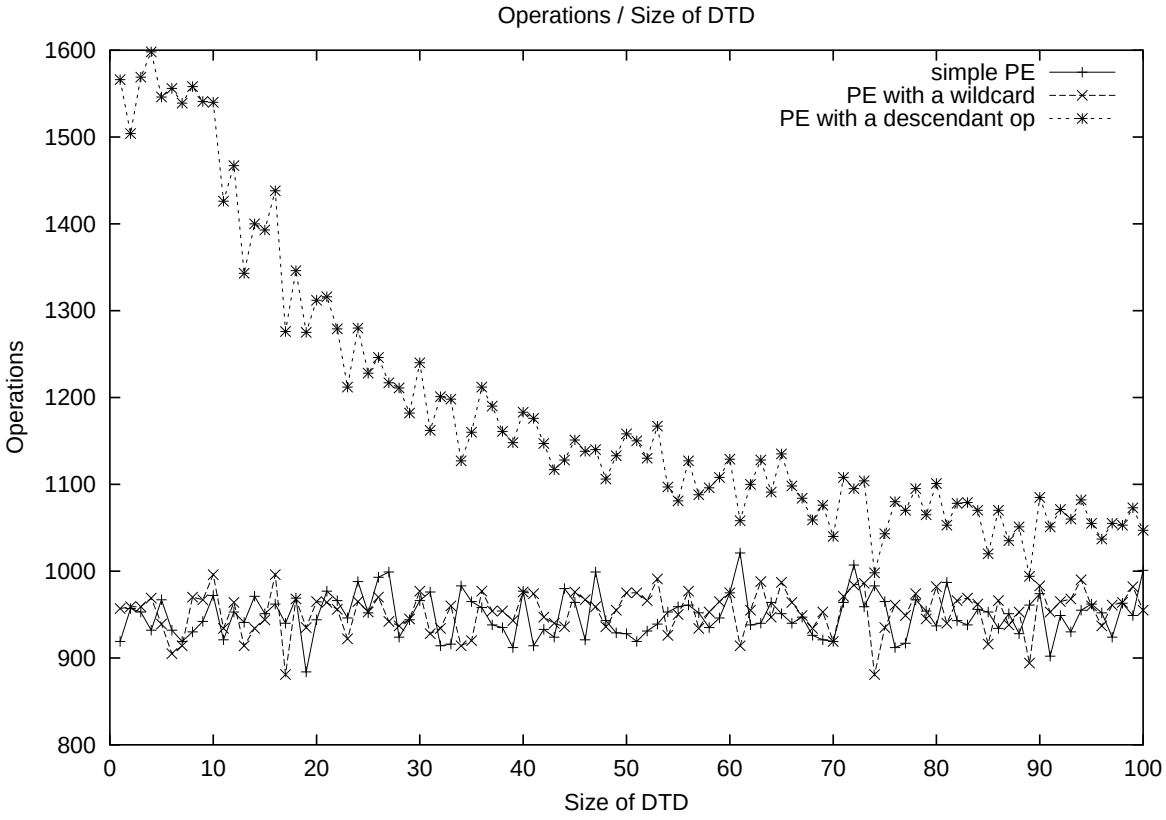


Figure 10: # of Operations / Size of DTD, simple traversal

`CLIENT-SEARCH-DOWN-ALL`, $|Q| = 1M$,$|I| = 20$,$Max(X_{\text{length}}) = 10$,$q_{\text{length}} = 3..5$,$u_{\text{length}} = 0..Max(X_{\text{length}}) - q_{\text{length}}$

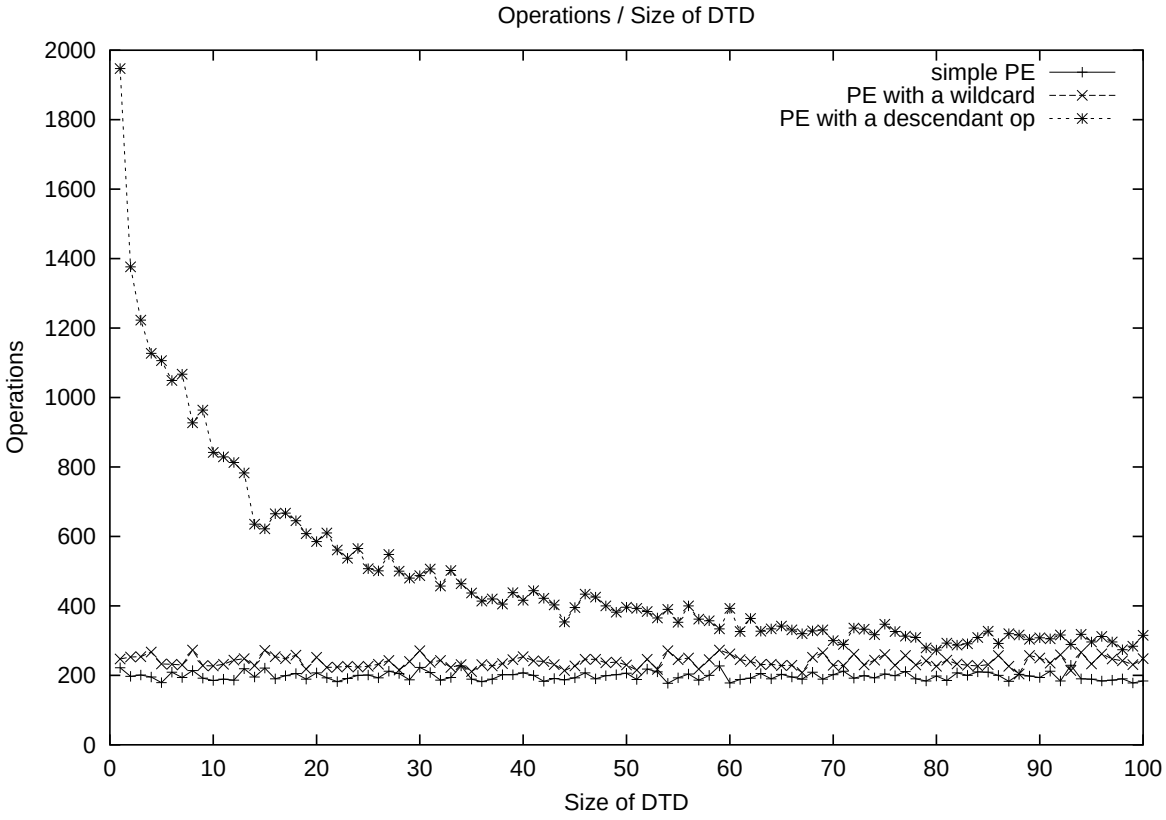


Figure 11: # of Operations / Size of DTD

CLIENT-SEARCH-DOWN, $|Q| = 1M, |I| = 20, Max(X_{\text{length}}) = 10, q_{\text{length}} = 3..5, u_{\text{length}} = 0..Max(X_{\text{length}}) - q_{\text{length}}$