On Efficient Matching of Streaming XML Documents and Queries

Laks V.S. Lakshmanan[1] and P. Sailaja[2]

[1] Univ. of British Columbia, Vancouver, BC V6T 1Z4, Canada
[2] Sun Microsystems, Bangalore, India
laks@cs.ubc.ca, Sailaja.Parthasarathy@sun.com

Abstract. Applications such as online shopping, e-commerce, and supply-chain management require the ability to manage large sets of specifications of products and/or services as well as of consumer requirements, and call for efficient matching of requirements to specifications.
Requirements are best viewed as "queries" and specifications as data, often represented in XML. We present a framework where requirements and specifications are both registered with and are maintained by a registry. On a periodical basis, the registry matches new incoming specifications, e.g., of products and services, against requirements, and notifies the owners of the requirements of matches found. This problem is dual to the conventional problem of database query processing in that the size of data (e.g., a document that is streaming by) is quite small compared to the number of registered queries (which can be very large). For performing matches efficiently, we propose the notion of a "requirements index", a notion that is dual to a traditional index. We provide efficient matching algorithms that use the proposed indexes. Our prototype MatchMaker system implementation uses our requirements index-based matching algorithms as a core and provides timely notification service to registered users. We illustrate the effectiveness and scalability of the techniques developed with a detailed set of experiments.

1 Introduction

There are several applications where entities (e.g., records, objects, or trees) stream through and it is required to quickly determine which users have potential interest in them based on their known set of requirements. Applications such as online shopping, e-commerce, and supply chain management involve a large number of users, products, and services. Establishment of links between them cannot just rely on the consumer browsing the online database of products and services available, because (i) the number of items can be large, (ii) the consumers can be processes as well, e.g., as in a supply chain, and (iii) a setting where products and services are "matched" to the interested consumers out of many, based on the consumer requirements, as the products arrive, will be more effective than a static model which assumes a fixed database of available items, from which consumers pick out what they want. As another example, the huge volume of information flowing through the internet has spurred the so-called *publish-and-subscribe* applications (e.g., see [6]) where data items are conveyed to users selectively based on their "subscriptions", or expressions of interest, stated as queries. At the heart of these applications is a system that we call a *registry* which records and maintains the requirements of all registered consumers. The registry should manage large sets of specifications of products and/or services (which may stream through it) as well as of consumer requirements. It monitors incoming data, efficiently detects the portions of the data that are relevant to various consumers, and dispatches the appropriate information to them.

Requirements are best viewed as "queries" and specifications as "data". On a periodical basis (which can be controlled), the registry matches new incoming specifications (e.g., of products and services) against registered requirements and notifies the owners of the requirements of matches found. In this paper, we assume all specifications are represented in XML. We adopt the now popular view that XML documents can be regarded as node-labeled trees.[1] For clarity and brevity, we depict XML documents in the form of such trees rather than use XML syntax. Figure 1(a) shows an example of a tree representing an XML document, where we have deliberately chosen a document with a DTD (not shown) permitting considerable flexibility. In the sequel, we call such trees *data trees*. We associate a number with each node of a data tree. Following a commonly used convention (e.g., see [4, 13]), we assume that node numbers are assigned by a preorder enumeration of nodes. Popular XML query languages such as XQL, Quilt, and XQuery employ a paradigm where information in an XML document is extracted by specifying patterns which are themselves trees. E.g., the XQuery query[2]

```
<Result>
FOR       $p IN sourceDB//part,
          $b IN $p/brand,
          $q IN $p//part
WHERE     A2D IN $q/name AND
          AMD IN $q/brand
RETURN    $p
</Result>
```

which asks for parts which have an associated brand and a subpart with name 'A2D' and brand 'AMD', corresponds to the tree pattern Figure 1(b), "**P**". In that figure, each node is labeled by a constant representing either the tag associated with an XML element (e.g., part) or a domain value (e.g., 'AMD') that is supposed to be found in the document queried. A unique node (root in the case of **P**) marked with a '*', identifies the "distinguished node", which corresponds to the element returned by the query. Solid edges represent direct element-subelement (i.e. parent-child) relationship and dotted edges represent transitive (i.e. ancestor-descendant) relationship. As another example, the XQL query `part/subpart[//'speakers']`, which finds all subparts of parts at any level of the document which have a subelement occurrence of 'speakers' at any level, corresponds to the tree pattern Figure 1(b), "**T**".

We call trees such as in Figure 1(b) *query trees*. Query trees are trees whose nodes are labeled by constants, and edges can be either of ancestor-descendant (ad) type or of parent-child (pc) type. There is a unique node (marked '*') in a query tree called the *distinguished node*.

---

[1] Cross-links corresponding to IDREFS are distinguished from edges corresponding to subelements for this purpose.

[2] The requirement/query is directed against a conceptual "sourceDB" that various specifications (documents) stream through.

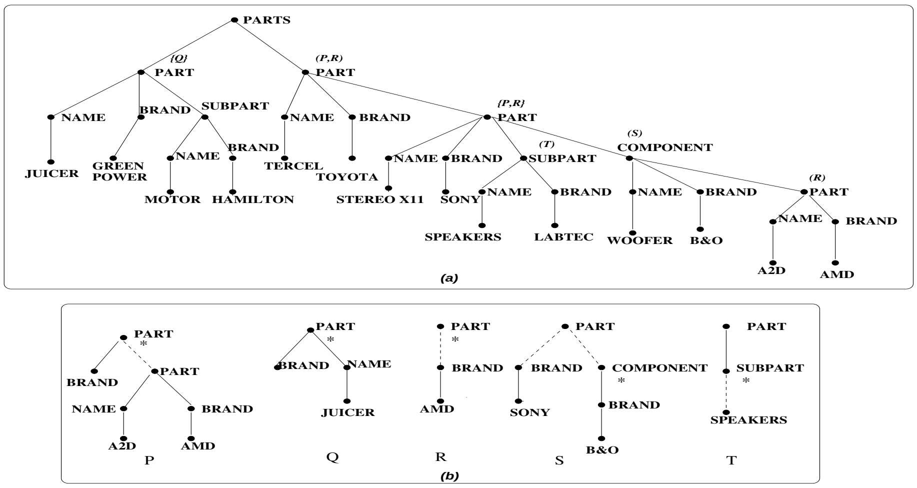

Fig. 1. The Labeling Problem Illustrated: (a)A data tree showing query labeling (node numbers omitted to avoid clutter); (b) A collection of queries (distinguished nodes marked '*').

We have implemented a MatchMaker system for matching XML documents to queries and for providing notification service. As an overview, XML data streams through the MatchMaker, with which users have registered their requirements in the form of queries, in a requirements registry. The MatchMaker consults the registry in determining which users a given data element is relevant to. It then dispatches the data elements relevant to different users. The architecture of MatchMaker as well as implementation details are discussed in the full version of this paper [14]. A prototype demo of MatchMaker is described in [15].

One of the key problems in realizing a MatchMaker is the *query labeling problem* (formally defined in Section 2 and illustrated below). Intuitively, given an XML document, for each element, we need to determine efficiently those queries that are answered by this element.

Example 11 [Query Labeling Problem Illustrated]
Suppose the user requirements correspond to the queries in Figure 1(b). Each user (i.e. query) is identified with a name. What we would like to do is identify for each element of the document of Figure 1(a), those users whose query is answered by it. For example, the right child of the root is associated with the set $\{P,R\}$, since precisely queries P,R in Figure 1(b) are answered by the subelement rooted at this node. Since a query may have multiple answers from one document, we need to determine not just *whether* a given document is relevant to a user/query, but also *which part* of the document a user is interested in. In Figure 1(a), each node is labeled with the set of queries that are answered by the element represented by the node.

A naive way to obtain these labels is to process the user queries, one at a time, finding all its matchings, and compile the answers into appropriate label sets for the document nodes. This strategy is very inefficient as it makes a number of passes over the given document, proportional to the number of queries. A more clever approach is to devise algorithms that make a *constant number of passes* over the document and determine the queries answered by each of its elements. This will permit set-oriented processing whereby multiple queries are processed together. Such an algorithm is non-trivial since: (i) queries may have repeating tags and (ii) the same query may have mutiple matchings into a given document. Both these features are illustrated in Figure 1. A main contribution of the paper is the development of algorithms which make a bounded number (usually two) of passes over documents to obtain a correct query labeling. ∎

It is worth noting that the problem of determining the set of queries answered by (the subtree rooted at) a given data tree node is *dual* to the usual problem of finding answers to a given query. The reason is that in the latter case, we consider answering one query against a large collection of data objects. By contrast, our problem is one of determining the set of queries (out of many) that are answered by one specific data object. *Thus, whereas scalability w.r.t. the database size is an important concern for the usual query answering problem, for the query labeling problem, it is scalability w.r.t. the number of queries that is the major concern.* Besides, given the streaming nature of our data objects, it is crucial that whatever algorithms we develop for query labeling make a *small number of passes* over the incoming data tree and efficiently determine all relevant queries.

We make the following specific contributions in this paper.

- We propose the notion of a "requirements index" for solving the query labeling problem efficiently. A requirements index is *dual* to the usual notion of index. We propose two different dual indexes (Section 3).
- Using the application of matching product requirements to product specifications as an example, we illustrate how our dual indexes work and provide efficient algorithms for query labeling (Sections 4 and 5). *Our algorithms make no more than two passes over an input XML document.*
- To evaluate our ideas and the labeling algorithms, we ran a detailed set of experiments (Section 6). Our results establish the effectiveness and scalability of our algorithms as well illustrate the tradeoffs involved.

Section 2 formalizes the problem studied. Section 7 presents related work, while 8 summarizes the paper and discusses future research. Proofs are suppressed for lack of space and can be found in [14].

2 The Problem

In this section, we describe the class of queries considered, formalize the notion of query answers, and give a precise statement of the problem addressed in the paper.

The Class of Queries Considered : As mentioned in the introduction, almost all known languages for querying XML data employ a basic paradigm of specifying a tree pattern and extracting all instances (matchings) of such a pattern in the XML database. In some cases, the pattern might simply be a chain, in others a general tree. The edges in the pattern may specify a simple parent-child (direct element-subelement) relationship or an ancestor-descendant (transitive element-subelement) relationship (see Section 1 for examples). Rather than consider arbitrary queries expressible in XML query languages, we consider the class of queries captured by query trees (formally defined in Section 1). We note that query trees abstract a large fragment of XPath expressions.

The semistructured nature of the data contributes to an inherent complexity. In addition, the query pattern specification allows ancestor-descendant edges. Efficiently matching a large collection of patterns containing such edges against a data tree is non-trivial.

Matchings and Answers : Answers to queries are found by matchings. Given a query tree Q and a data tree T, a *matching* of Q into T is a mapping h from the nodes of Q to those of T such that: (i) for every non-leaf node x in Q, the tag that labels x in Q is identical to the tag of $h(x)$ in T; (ii) for every leaf x in Q, either the label of x is identical to the tag of $h(x)$ or the label of x is identical to the domain value at the node $h(x)$ in T;[3] (iii) whenever (x,y) is a pc (resp., ad) edge in Q, $h(y)$ is a child (resp., (proper) descendant) of $h(x)$ in T. For a given matching h of Q into T, the corresponding answer is the subtree of T rooted at $h(x_d)$, where x_d is the distinguished node of the query tree Q. In this case, we call $h(x_d)$ the *answer node* associated with the matching h. A query may have more than one matching into a data tree and corresponding answers.

User queries are specified using query trees, optionally with additional order predicates. An order predicate is of the form $u \prec v$, where u, v are node labels in a query tree, and $\prec$ says u must precede v in any matching. As an example, in Figure 1(b), query Q, we might wish to state that the element BRAND appears before the NAME element. This may be accomplished by adding in the predicate BRAND $\prec$ NAME.[4] The notion of matching extends in a straightforward way to handle such order predicates. Specifically, a predicate $x \prec y$ is satisfied by a matching h provided the node number associated with $h(x)$ is less than that associated with $h(y)$. Decoupling query trees from order is valuable and affords flexibility in imposing order in a selective way, based on user requirements. For example, a user may not care about the relative order between author list and publisher, but may consider the order of authors important.

Problem Statement : The problem is to label each node of the document tree with the list of queries that are answered by the subtree rooted at the node. We formalize this notion as follows. We begin with query trees which are chains. We call them chain queries.

1. The Chain Labeling Problem: Let T be a data tree and let $Q_1, \ldots, Q_n$ be chain queries. Then label each node of T with the list of queries such that query Q_i belongs to the label of node u if and only if there is a matching h of Q_i into T such that h maps the distinguished node of Q_i to u.

The importance of considering chain queries is two-fold: firstly, they frequently arise in practice and as such, represent a basic class of queries; secondly, we will derive one of our algorithms for the tree labeling problem below by building on the algorithm for chain labeling.

[3] Instead of exact identity, we could have similarity or substring occurrence.

[4] A more rigorous way is to associate a unique number with each query tree node and use these numbers in place of node labels, as in $x \prec y$.

2. The Tree Labeling Problem: The problem is the same as the chain labeling problem, except the queries $Q_1, \ldots, Q_n$ are arbitrary query trees.

3 Dual Index

The "requirements index" is dual to the conventional index, so we call it the *dual index*: given an XML document in which a certain tag, a domain value, or a parent-child pair, ancestor-descendant pair, or some such "pattern" appears, it helps quickly determine the queries to which this pattern is relevant. Depending on the labeling algorithm adopted, the nature of the dual index may vary.

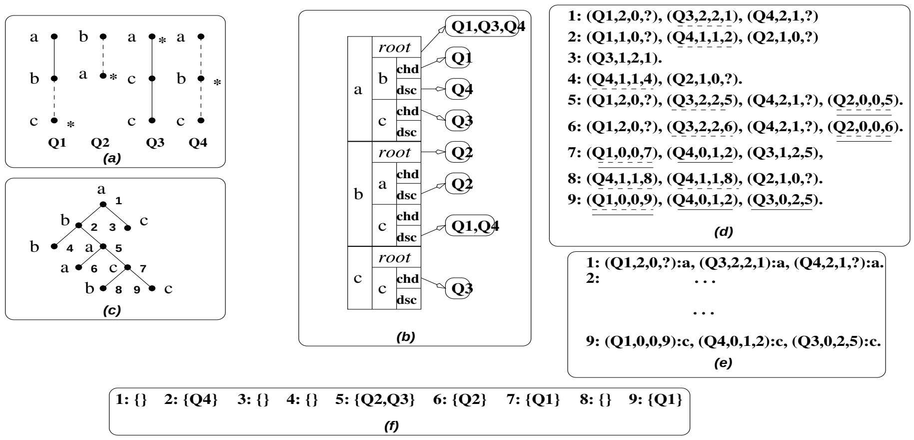

Fig. 2. Chain Dual Index: (a) queries; (b) dual index (lists not shown are empty; *chd* = *child*, *dsc* = *desc*); (c) a sample document; (d) CML lists; (e) PL lists: for brevity, only shown nodes 1 and 9; besides, only those entries of $PL(9)$ not "inherited" from anestors of 9 are shown; (f) QL lists.

For chain labeling, the basic questions that need to be answered quickly are: (i) given a constant t, which queries contain an occurrence of t as a root?[5] (ii) given a pair of constants t_i, t_j, in which queries do they appear as a parent-child pair, and in which queries as an ancestor-descendant pair?[6] Notice that an ancestor-descendant pair (t_i, t_j) in a query Q means that t_j is an ad-child of t_i in Q. We call an access structure that supports efficient lookups of this form *chain dual index*. In our chain labeling algorithm, we denote lookup operations as DualIndex$[t](root)$, DualIndex$[t_i][t_j](child)$, or DualIndex$[t_i][t_j](desc)$, which respectively return the list of queries where t appears as the root tag, (t_i, t_j) appears as the tags/constants of a parent-child pair, and of an ancestor-descendant pair. E.g., Figure 2 (a)-(b) shows a sample chain dual index: e.g., DualIndex$[a](root) = \{Q1, Q3, Q4\}$, DualIndex$[a][b](child) = \{Q1\}$, and DualIndex$[a][b](desc) = \{Q4\}$.

For tree labeling, the questions that need to be answered quickly are: (i) given a constant, which are the queries in which it occurs as a leaf and the node numbers of the occurrences, and what are the node numbers of the distinguished node of these queries, and (ii) given a tag, for each query in which it has a non-leaf occurrence, what

[5] So, t must be a tag.

[6] Here, t_i must be a tag, while t_j may be a tag or a domain value.

are the node numbers of its pc-children and the node numbers of its ad-children? We call an access structure that supports efficient lookups of this form *tree dual index*. Figure 3(a)-(b) shows some queries and their associated tree dual index. In our tree labeling algorithm, we denote the above lookups as DualIndex$[t](L)$ and DualIndex$[t](N)$ respectively. Each entry in DualIndex$[t](L)$ is of the form (P,l,S), where P is a query, l is the number of its distinguished node, and S is the set of numbers of leaves of P with constant t. Each entry in DualIndex$[t](N)$ is of the form (P,i,B,S,S'), where P is a query, i is the number a node in P with constant t, and S,S' are respectively the sets of numbers of i's pc-children and ad-children; finally, B is a boolean which says whether node i has an ad-parent in P (True) or not (False). The motivation for this additional bit will be clear in Section 5. Figure 3(b) illustrates the tree dual index for the sample set of queries in Figure 3(a). E.g., DualIndex$[a](N)$ contains the entries $(P,1,F,\{2,3\},\{\})$ and $(P,4,T,\{6\},\{5\})$. This is because a occurs at nodes 1 and 4 in query P. Of these, the first occurrence has nodes $2,3$ as its pc-children and no ad-children, while the second has pc-child 6 and ad-child 5.

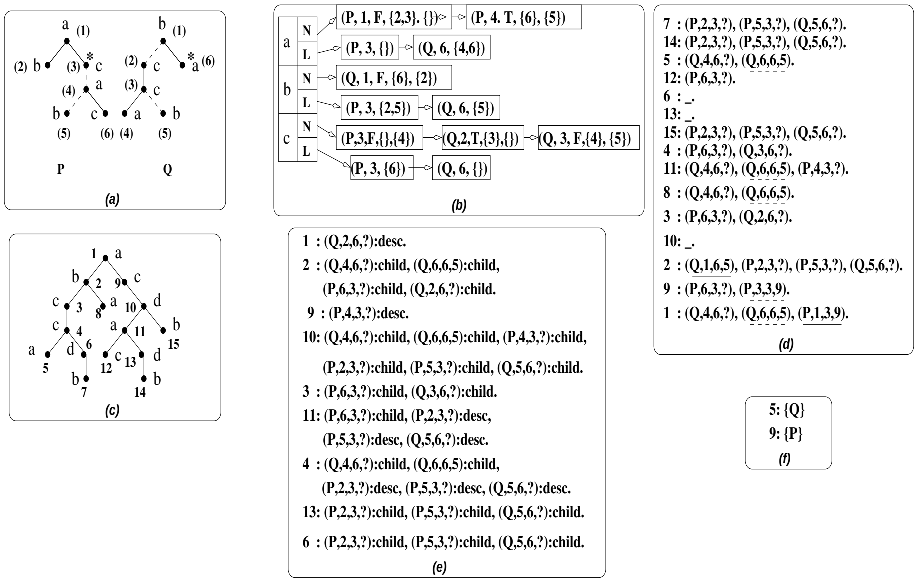

Fig. 3. Tree Dual Index: (a) queries; (b) dual index; (c) sample document; (d)TML; (e)PL; (f) QL.

4 Chain Queries

In this section, we present an algorithm called *Algorithm Node-Centric-Chain* for chain labeling. Given a data tree, we seek to label each of its nodes with the list of chain queries that are answered by it. The main challenge is to do this with a small number of passes over the data tree, regardless of the number of queries, processing many queries at once. Figure 4 gives an algorithm for chain labeling. *For expository reasons, in our presentation of query labeling algorithms in this and the next section, we associate several lists with each data tree node. We stress that in our implementation we compress these lists into a single one for reasons of efficiency [14].*

With each data tree node, we associate three lists: the QL (for query labeling) list, which will eventually contain those queries answered by the (subtree rooted at the) node, a list called CML (for chain matching list) that tracks which queries have so far been matched and how far, and an auxiliary list called PL (for push list) that is necessary to manage CML. Each entry in the CML of a node u will be of the form (P,i,j,x), where P is a query, and i,j are non-negative integers, and x is either '?' or is an integer. Such an entry signifies that there is a partial matching of the part of the chain P from its root down to the node whose distance from leaf is i, and j is the distance between the distinguished node and the leaf. In addition, when x is an integer, it says the data tree node with number x is the answer node corresponding to this matching. E.g., in Figure 2(d), for node 1 of the document, one of the CML entries is $(Q1,2,0,?)$, signifying that in query $Q1$, the part that includes only the query root (tag a) can be partially matched into node 1. The number 2 says the matched query node is at a distance[7] 2 from the query leaf, and the number 0 says the distinguished node is at distance 0 from the query leaf, i.e. it *is* the leaf. The '?' says the location of the answer node for this matching is unknown at this point. Note that the CML associated with a data tree node may have multiple entries corresponding to the same query, signifying multiple matchings. Entries in PL are of the form $(P,l,m,x):t$, where P,l,m,x are as above and t is a constant (tag). Such an entry belongs to the PL of a node u exactly when some ancestor of u in the document has tag t and the part of P from root down to the node with distance l to the leaf has been matched to this ancestor, and x has the same interpretation as above, i.e. the ancestor's CML contains (P,l,m,x).

An entry (P,l,m,x) can get into the CML of a data tree node u in one of two ways: (1) The root of P has tag identical to $tag(u)$. In this case, l must be the height of P and m the distance from leaf to distinguished node. If $l=m$, we know the location of the answer node, viz., node u. (2) The CML of u's parent in the document, say v, contains the entry $(P,l+1,m,x)$ and $(tag(v),tag(u))$ appears either as a pc-edge or as an ad-edge in query P. Note that the latter check can be performed by looking up the chain dual index via DualIndex$[tag(v)][tag(u)](rel)$, where *rel* is either *child* or *desc*. It is possible that an ancestor (which is not the parent) v of u contains the entry $(P,l+1,m,x)$ instead. This should cause (P,l,m,x) to be added to the CML of u, where x should be set to the number of node u if $l=m$, and left as '?' otherwise. Instead of searching ancestors of u (which will make the number of passes large), we look for an entry of the form $(P,l+1,m,x):t$ in the PL of u, such that $(t,tag(u))$ appears as an ad-edge in query P. Here, t denotes the ancestor's tag. This keeps the number of passes constant.

Next, the PL is maintained as follows. Whenever an entry (P,l,m,x) is added to the CML of a node u, the entry $(P,l,m,x):tag(u)$ is added to the PL of u. Whenever x is set to a node number in a CML entry, the component x in the corresponding PL entry is also replaced. The contents of the PL of a node are propagated to its children. The following theorem shows there is an algorithm for chain labeling that makes two passes over the data tree. It assumes the existence of a chain dual index. Construction of dual indexes is discussed in [14].

As an example, in Figure 2(c), the CML of node 1 will initially get the entries $(Q1,2,0,?)$, $(Q3,2,2,1)$, $(Q4,2,1,?)$. In the second entry, the last component 1 says

[7] Solid and dotted edges are counted alike for this purpose.

Algorithm Node-Centric-Chain
Input: User queries $Q_1, ..., Q_n$;
Node labeled data tree T;
For simplicity, we use u also as the preorder rank associated with node u;
Output: A query labeling of T;
1. Initialize the lists CML, PL, and QL of every node to empty lists;
2. Traverse nodes of T in preorder;
  for each node u {
2.1. for every ($Q \in$ DualIndex$[tag(u)](root)$) {
    add $(Q,h,d,?)$ to $CML(u)$, where h is the height of Q and
    d is the distance of distinguished node from leaf;
    add $(Q,h,d,?) : tag(u)$ to $PL(u)$;
    if $(h = d)$ replace '?' by u;
    //u is then a node to which the distinguished node of Q is matched. }
2.2. if (u has parent) {
    let v be the parent;
2.2.1. for every ($Q \in$ DualIndex$[tag(v)][tag(u)](desc) \cup$
      DualIndex$[tag(v)][tag(u)](child)$) {
      if $(\exists i, j, x :\ (Q, i+1, j, x) \in CML(v))$ {
        add (Q,i,j,x) to $CML(u)$ and $tag(u) : (Q,i,j,x)$ to $PL(u)$;
        if $(x =$ '?'$)$ { if $(i = j)$ replace x by u; } } }
2.2.2. for every $((Q, i+1, j, x) : t \in PL(v))$ {
      if $(Q \in DualIndex[t][tag(u)](desc))$ {
        add (Q,i,j,x) to $CML(u)$ and $tag(u) : (Q,i,j,x)$ to $PL(u)$;
        if $(x =$ '?'$)$ { if $(i = j)$ replace x by u; } } }
2.2.3. for every $((P,m,n,x) : t \in PL(v))$ add $(P,m,n,x) : t$ to $PL(u)$; } }
3. Traverse T postorder;
  for each node u {
3.1. for (every $(P, 0, j, x) \in CML(u)$) {
    add P to $QL(x)$; } }

Fig. 4. Labeling Chain Queries: The Node-Centric Way

node 1 in the document is the answer node (corresponding to that matching). Figure 2(d)-(e) shows the CML and PL lists computed at the end of step 2 of the algorithm. Figure 2(f) shows the QL lists computed at the end of step 3. Note that the entry $(Q4, 1, 1, 8)$ appears twice in $CML(8)$. They are inserted by two different matchings, one that involves node 1 and another that involves node 5. In the figure, entries underlined by a dotted line indicate the first time (in step 2) when the component is replaced from a '?' to a node number for those entries. Entries underlined by a solid line indicate successful completion of matchings (as indicated by the third component being zero).

Theorem 1. (Chain Labeling) : Let T be a data tree and $Q_1, ..., Q_n$ any set of chain queries. There is an algorithm that correctly obtains the query labeling of every node of T. The algorithm makes no more than two passes over T. Furthermore, the number of I/O invocations of the algorithm is no more than $n * (2 + p)$, where n is the number of nodes in the document tree and p is the average size of the node PL lists associated with data tree nodes. ▮

5 Tree Queries

In this section, we first discuss a direct bottom-up algorithm called *Algorithm Node-Centric-Trees* for tree labeling. As will be shown later, this algorithm makes just two passes over the data tree T and correctly obtains its query labeling w.r.t. a given set of queries which are general trees.

5.1 A Direct Algorithm

Figure 5 shows Algorithm *Algorithm Node-Centric-Trees*. As with the chain labeling algorithm, we associate several lists with each node u in T. (1) The Tree Matching List (TML) contains entries of the form (P,l,m,x) where P is a query, l the number of a node in P, m the number of the distinguished node of P, and x is either '?' or the number of the node in T to which the distinguished node of P has been matched. (2) The push list (PL) contains entries of the form $(P,l,m,x) : rel$ where P,l,m,x are as above and *rel* is either *child* or *desc*. It says there is a child or descendant (depending on *rel*) of the current node u whose TML contains the entry (P,l,m,x). (3) The query labeling list (QL) contains the list of queries for which u is an answer node.

These lists are managed as follows. Leaf entries in the tree dual index cause entries to be inserted into the TML lists of nodes. More precisely, let the current node u have tag t, and let DualIndex$[t](L)$ contain the entry (P,m,S). In this case, for every node $l \in S$, we insert the entry $(P,l,m,?)$ into $TML(u)$; in case $l = m$, we replace '?' by the node number u. This signifies that the distinguished node of P has been matched into node u of T. Inductively, let (P,l,B,C,D) be an entry in DualIndex$[tag(u)](N)$. If according to $PL(u)$, for every $c \in C$, there is a child of u into which the subtree of P rooted at c can be successfully matched, and for every $d \in D$, there is a descendant of u into which the subtree of P rooted at d can be matched, then we add the entry (P,l,m,x) to $TML(u)$. Here, the value of m is obtained from the entries in $PL(u)$ that were used to make this decision. If at least one of the relevant $PL(u)$ entries contains a value for x other than '?', we use that value in the newly added entry in $TML(u)$; otherwise, we set x to '?' in $TML(u)$.

The PL list is first fed by entries added to TML: for every entry (P,l,m,x) added to $TML(u)$, we add the entry $(P,l,m,x) : child$ to $PL(parent(u))$. Secondly, whenever $PL(u)$ contains the entry $(P,l,m,x) : rel$, we add the entry $(P,l,m,x) : desc$ to $PL(parent(u))$. This latter step can result in an explosion in the size of the PL lists as every entry added to the PL of a node contributes entries to all its ancestors' PL lists. This can be pruned as follows. Suppose $PL(u)$ contains (P,l,m,x) and suppose node l of query P has no ad-parent, i.e. either it has no parent or it has a pc-parent. In the former case, there is no need to "propagate" the PL entry from u to its parent. In the latter case, since the propagated entry would be $(P,l,m,x) : desc$, it cannot contribute to the addition of any entry to $PL(parent(u))$.

As an example, in Figure 3(c), there is a sample document. The figure illustrates how its query labeling is constructed by Algorithm Node-Centric-Trees w.r.t. the queries of Figure 3(a), including the details of the TML, PL, and QL lists. For instance, $(P,2,3,?)$ is added to $TML(7)$ since $tag(7) = b$ and $(P,3,\{2,5\})$ belongs to DualIndex$[b](L)$. As

Algorithm Node-Centric-Trees

Input: User queries $Q_1,\ldots,Q_n$; Node labeled data tree T;
Output: A query labeling of T;
1. let h = height of T;
2. for ($i = h$; $i \geq 0$; $i--$) {
2.1 for (each node u at level i) {
2.1.1. for (each (P, m, S) in DualIndex$[tag(u)](L)$) {
for (each l in S) {
add $(P, l, m, ?)$ to $TML(u)$; //base case entries
if $(l = m)$ replace '?' by u; } }
2.1.2. for (each entry e in DualIndex$[tag(u)](N)$) {
2.1.2.1. let e be (P, l, B, C, D);
2.1.2.2. if (not B) $mask(P, l) = true$;
//if u has no ad-parent, mask propagation of PL entries.
2.1.2.3. let $R = \{(m, x) \mid [\forall ci \in C : (P, ci, m, y) : child \in PL(u)\&$
$((y = x) \vee (y = \text{'?'}))]\ \&\ \ [\forall di \in D : ((P, ci, m, y) : child \in PL(u) \vee$
$(P, ci, m, y) : desc \in PL(u))\&((y = x) \vee (y = \text{'?'}))]\}$;
2.1.2.4. if $(R \neq \emptyset)$ {
for (each (m, x) in R) {
add (P, l, m, x) to $TML(u)$;
if $(l = m)$ replace x by u; } } }
2.1.3 for (each entry (P, l, m, x) in $TML(u)$) {
2.1.3.1. add $(P, l, m, x) : child$ to $PL(parent(u))$;
//Push that entry to the parent's pushlist
//indicating that this corresp. to a child edge. }
2.1.4. for (each entry $(P, l, m, x) : rel$ in $PL(u)$) {
if (not $mask(P, l)$)
add $(P, l, m, x) : desc$ to $PL(parent(u))$; } }
3. Traverse T preorder;
for each node u of T {
if ((P, l, l, x) is in $TML(u)$) add P to $QL(u)$; }

Fig. 5. Labeling Tree Queries: The Node-Centric Way

an example of the inductive case for adding entries into TML, consider the addition of $(Q, 3, 6, ?)$ into $TML(4)$. This is justified by the presence of $(Q, 4, 6, ?) : child$ and $(Q, 5, 6, ?) : desc$ in $PL(4)$. The PL entries can be similarly explained. In the figure, those TML entries where the last component x gets defined for the first time are underlined in dotted line, whereas those entries which signify a complete matching (as indicated by $l = 1$) are underlined in solid line.

Theorem 2. (Tree Labeling) : Let T be a data tree and $Q_1,\ldots,Q_n$ any set of query trees. There is an algorithm that correctly obtains the query labeling of every node of T. The algorithm makes no more than two passes over T. Furthermore, the number of I/O invocations of the algorithm is at most $2 * n$, where n is the number of nodes of the document tree. ∎

5.2 The Chain-Split Algorithm

In this section, we develop an approach for tree labeling that builds on the previously developed approach for chain labeling. The main motivation of this approach is based on the assumption that matching chains is easier than matching arbitrary trees, so if a query contains many chains, it makes sense to exploit them. A *chain* in a query tree P is any path $(x_1,...,x_k)$ such that every node $x_i, 1 < i < k$, has outdegree 1 in P.[8] Such a chain is maximal provided: (i) either x_k is a leaf of P, or it has outdegree > 1 and (ii) either x_1 is the root of P, or it has outdegree > 1. In the sequel, by chains we mean maximal chains. For example, consider Figure 6(a). It shows four chains in query P – $P1 = (1,2), P2 = (1,3,4), P3 = (4,5), P4 = (4,6)$; similarly, there are four chains in query Q – $Q1, Q2, Q3, Q4$. Suppose we split given queries into chains and match them into the data tree. Then, we should be able to make use of the information collected about the various matchings of chains in matching the original queries themselves efficiently.

For brevity, we only outline the main steps of the Chain-Split algorithm for tree labeling (instead of a formal description). We will need the following notions. A node of a query tree P is a *junction node* if it is either the root, or a leaf, or has outdegree > 1. In the chain split algorithm, we first match all the chains obtained from query trees using a chain dual index. In addition, we use an auxiliary data structure called *junction index*. The junction index is indexed by constants, just like the dual index. The index for any tag consists of entries of the form $(P,n) : \ (Pi,l_i),...,(Pj,l_j)$, where P is a query, $Pi,...,Pj$ are its chains, and $n,l_i,...,l_j$ are node numbers in P. It says *if* a node u in a data tree satisfies the following conditions: (i) the chains $Pi,...,Pj$ are matched into the data tree and u is the match of all their roots; (ii) $v_i,...,v_j$ are the corresponding matches of their leaves[9]; (iii) the subtree of P rooted at l_i (resp., $...,l_j$) is matched into the subtree of the data tree rooted at v_i (resp., $...,v_j$); *then* one can conclude that the subtree of P rooted at n is matched into the subtree rooted at u. In this sense, a junction index entry is like a rule with a consequent (P,n) and an antecedent$(Pi,l_i),...,(Pj,l_j)$. Entries may have an empty antecedent. Note that the junction index may contain more than one entry for a given tag and a given query.

Figure 6 illustrates how this algorithm works. A detailed exposition is suppressed for lack of space and can be found in [14].

A formal presentation of the chain split algorithm, omitted here for brevity, can be easily formulated from this exposition. A theorem similar to Theorem 2 can be proved for this algorithm as well.

6 Experiments

We conducted a series of experiments to evaluate the effectiveness of the labeling algorithms developed. The algorithms presented were rigorously tested under different workloads.

Experimental setup: Our MatchMaker system is implemented using JDK1.3, GnuC++2.96, and BerkeleyDB3.17 [2]. The requirements indexes, that we proposed are all disk resident and all other data structures are memory resident. The experiments were conducted

[8] Note that all paths in our trees are downward.

[9] They need *not* be P's leaves.

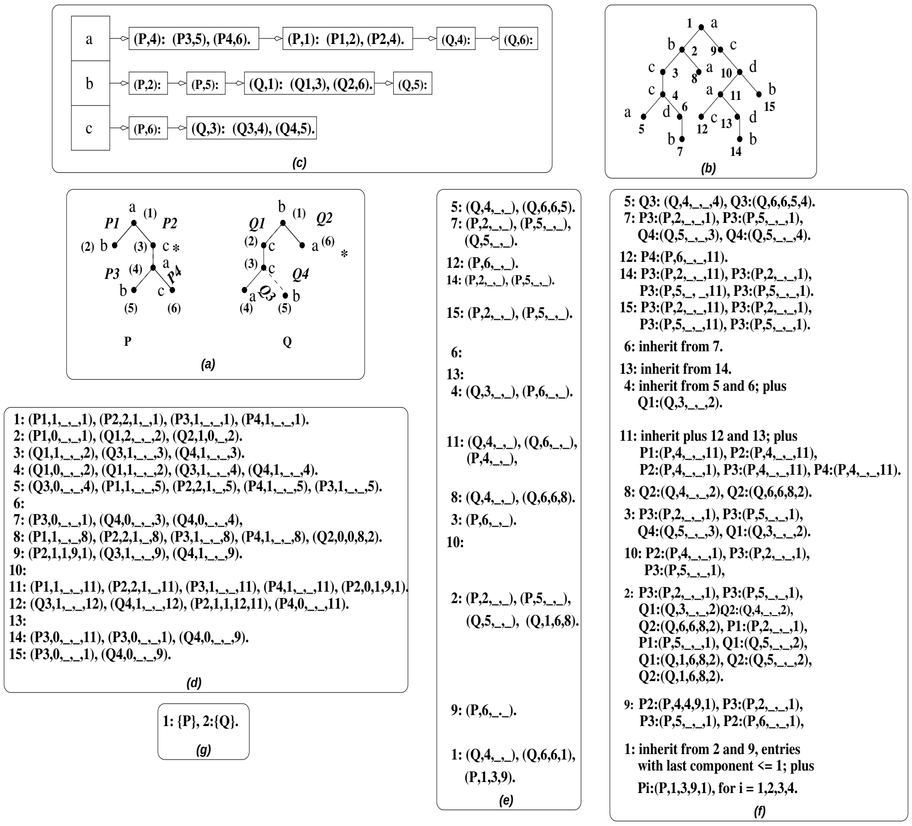

Fig. 6. Chain Split Algorithm Illustrated: (a) Queries; (b) Document; (c) Junction Index; (d) CML lists for the split chains; (e) TML lists corresponding to the queries; (f) PropList; (g) QL lists.

on a Intel Pentium III, dual processor with 1GB RAM, and 512KB cache operating on RedHat Linux 7.0. We ran our bunch of experiments using the GedML DTD. It is a DTD for encoding genealogical datasets in XML, based on GEDCOM which is a widely used data format for genealogical data [9]. This DTD has about 120 elements. For analyzing our algorithms, we generated both queries and documents. For generating documents, we made use of IBM's XML Generator tool [5], that generates random instances of valid XML documents from a DTD and a set of constraints. We analyzed the algorithms for various kinds of workloads.

Results: The results that we describe in this section pertain to documents with 120 nodes. We conducted an extensive set of experiments by varying the length of chain queries (or depth for tree queries), the distance of distinguished node from the leaf node, and the results we furnish here are averaged over 3 runs. The document depth was set to 10 while the average fanout (for non-leaves) was randomly chosen between 2 and 5. The results are summarized in Figures **A-H**. For the sake of comparison, we implemented the query-at-a-time approach for query labeling. However, we found that for tree queries, this approach took too much time and hence we only show its performance for chain queries.

Chain Queries: For chain queries, we determined the time needed to label the document by varying the number of queries. We compared the time taken with a query-at-a-time approach for query labeling. The number of queries was varied from 10,000 to 100,000 and the time taken for labeling the document was measured. The queries were generated with uniform distribution as well as with skewed distribution. In both cases, the chain length was randomly chosen in the range 2 to 9 and the chain tags were chosen from a uniform distribution over the DTD tags. For the skew distribution, tags were chosen using a Zipf distribution. The queries generated may have multiple matchings as well. Figure 8:A depicts the results of this experiment. For the skew distribution, we chose a case where there was a 100% hit ratio (= 1 in the graph), meaning every query had one or more matchings into the document. The figure shows a speedup of about 5 for the chain labeling algorithm compared with the query-at-a-time approach. Figure 8:B shows the memory usage of the chain labeling algorithm under both distributions. Both figures show that the algorithm scales gracefully. For 100,000 queries time is about 100 sec and memory consumed is about 300MB.

Effect of Skewness: For analyzing the effect of skewness of the queries on the total time taken for labeling the document, we ran a set of experiments by fixing the number of queries to 20,000 and controlling the skew using Zipf distribution. In interpreting the results, we found it more informative to use the hit ratio defined above than to use the actual value of the skewness factor. As can be seen from Figure 8:C, as the ratio increases from 0 (no queries have a matching) to 1 (all of them have at least one matching), the chain labeling algorithm has a performance that scales gracefully, whereas the query-at-a-time approach degrades quickly. The reason is that as more queries start to have (matchings) answeres, the amount of work done by this approach increases dramatically. On the other hand, for the chain labeling algorithm, the amount of work is only indirectly influenced by the hit ratio, since this is determined by the number of document nodes and the length of the associated lists.

Tree Queries: As mentioned in the beginning of this section, we were unable to obtain the performance results for the query-at-a-time approach for tree queries in a reasonable amount of time. Consequently, our goal here was to compare the relative performance of the direct (bottom-up) tree labeling algorithm and the chain split algorithm and understand the tradeoffs. First, we focused on a realm where the number of junction nodes is a large proportion of the total number of nodes, for queries in the distribution. As expected, the direct algorithm performs better than chain split for this case. Figures 8:D, E, F show this. The memory usage for the algorithms is comparable. The figures show the performance for both uniform distribution and skewed distribution. As for chain queries, for skew, we used the Zipf distribution. The hit ratio used for the skew distribution case is 1. With respect to labeling time, the bottom-up algorithm is about 1.25-1.5 times better than the split chain algorithm.

Bottom-up vs. Split Chain Tradeoff: When the junction nodes form a small proportion of the total number of nodes, we expect that the split chain algorithm would take advantage of the ease of labeling chains. The additional overhead of matching the tree (using junction index) may well be offset by the speedup gained because of chain decomposition. To test this conjecture, we ran a set of experiments. By fixing the ratio of the number of junction nodes and the total number of nodes in a query to 0.125, we generated tree queries. Once again, we generated both uniform and Zipf distribution (with hit ratio 1) for queries. As shown by Figure 8:G, split chain can be about 1.4 faster than the bottom-up algorithm under these circumstances.

Summary: In sum, the experiments reveal that query labeling algorithms easily outperform query-at-a-time approach. In many cases (especially for tree labeling), query-at-a-time does not even finish in a reasonable amount of time, whereas the labeling algorithms show a performance that scales well with number of queries and skew factor. In the realm of tree labeling, the experiments show that direct bottom-up and chain split algorithms may be preferable under different conditions. The tradeoff is mainly driven by the ratio of junction nodes to the total number of nodes.

7 Related Work

The Tapestry mail management system [18], developed at Xerox Park, supports "continuous queries" against append-only databases, and provides perodical notification. Gupta and Srivastava [10] study the problem of view maintenance for a warehouse of newsgroups. The TriggerMan [11] system at the University of Florida is a scalable system for supporting millions of triggers in a DBMS context, using the notion of "expression signature", a common structure shared by many triggers. The NiagaraCQ system at Wisconsin [4] extends continuous queries to XML, and adapts the notion of expression signature for grouping XML-QL queries.

More recently, Fabret et al. [6, 7] describe a publish-and-subscribe system that maintains long-term subscription queries and notifies their owners whenever an incoming event matches the subscription, using a main memory algorithm with sophisticated caching policies. Subscription predicates are restricted to conjunctions of *attribute relOp value*. A direct adaptation of this approach for XML would not take advantage of the tree structure of documents and queries. The query subscription [8] project at Stanford has focused on merging overlapping queries to save on query compute time as well as broadcast time. The MatchMaker approach presented in this paper is fundamentally different from all the above in that it can be regarded as the dual of conventional query processing.

The XFilter project at Berkeley [1] is perhaps the closest to our work and is perhaps the first paper to adopt a dual approach to matching of XML documents to queries. There are important differences with our work. Firstly, they merely determine whether a document is relevant to a query (i.e. *contains* an answer to a query) as opposed to locating the answer element(s). Besides, they do not address multiple answers to a query within one document. As pointed out earlier, these two issues alone raise a major challenge for efficient matching, and we deal with both. Our formulation of the matching problem in terms of query labeling, as well as our matching algorithms, to the best of our knowledge, are novel. Also, we can formally show the correctness of our algorithms [14]. The problem of matching documents to queries (sometimes called user profiles) has been studied in the IR community. A critique of their approaches and a comparison with the database approach to matching appears in [1]. Intel's XML Accelerator [12] is an industry product providing XML document filtering.

Finally, in very recent work, Chan et al. at Bell Labs [3] develop a trie-based index, called XTrie, for indexing a large collection of subscription queries, stated as XPath expressions. The intuitive idea is to split an XPath tree into maximal paths free of ad edges, called substrings, and index them using a trie. The resulting XTrie index, together with their matching algorithm, makes it possible to reduce unnecessary index probes and avoid redundant matchings. Just like the XFilter work, the authors are mainly interested in finding queries which have at least one matching (anywhere) in a given document, which distinguishes our work. The notion of substring decomposition of [3] is similar in spirit to, but is technically different from, our chain decomposition of tree queries. Since these two ideas are orthogonal, it suggests they can be combined to take advantage of the best of both worlds. In particular, this makes it possible to extend the XTrie approach not only for determining existence of matchings, but to actually extract the answers and do so efficiently. Nguyen et al. [17] describe a system for efficient monitoring of changes to XML documents flowing through a warehouse, and as such the concerns are somewhat different.

8 Summary and Future Work

Motivated by applications requiring efficient matching of specifications to requirements, both in XML, we proposed the problem of query labeling. We presented efficient algorithms for query labeling for chain as well as tree queries. The algorithms can be formally shown to be correct. Our experimental results show the effectiveness of our labeling algorithms. As discussed earlier, the algorithms, by virtue of handling streaming XML documents, are also applicable for publish-and-subscribe applications on the internet, involving XML documents and queries.

Like the XFilter and XTrie papers, this paper addresses the first steps in developing large scale publish-and-subscribe systems involving streaming XML data. Specifically, our goal is to extend MatchMaker so as to cater to subscriptions covering larger fragments of XML query languages. The recently proposed TAX algebra for XML [13] offers a convenient framework with which to proceed.

We plan to do detailed experiments comparing the performance of our labeling algorithms and index structures with the XTrie approach. In addition, it would also be fruitful to determine how ideas from the two works can be combined to lead to even more powerful algorithms.

Maintaining dual indexes in an incremental fashion is an important issue. Integration of database style matching of documents to queries with keyword search and handling of the concomitant uncertainty is another interesting direction. Our ongoing work addresses some of these issues.

Acknowledgements

Lakshmanan's work was supported in part by grants from NSERC, NCE/IRIS/NSERC, and by a seed grant from IIT-Bombay. Sailaja's research was done while she was at IIT-Bombay and was supported by a Ramarao V. Nilekani Fellowship.

References

1. Mehmet Altinel, Michael J.Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. VLDB*, 2000.
2. Berkeley DB Database System. Downloadable from `http://www.sleepycat.com/`.

3. Chee-Yong Chan , Pascal Felber, Minos Garofalakis, and Rajeev Rastogi. Efficient Filtering of XML Documents with XPath Expressions. *Proc. ICDE*, San Jose, CA, Feb. 2002. To appear.
4. J.Chen, D.DeWitt, F.Tian, and Y.Wang. NiagaraCQ: A scalable continuous query System for Internet Dtatabases. In *ACM SIGMOD*, May 2000.
5. A.L. Diaz and D. Lovell. XML Generator. `http://www.alphaworks/ibm.com/tech/xmlgnerator`, Sept. 1999.
6. Franoise Fabret, Hans-Arno Jacobsen, Franois LLirbat, Joo Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe. In *ACM SIGMOD*, May 2001.
7. F.B Fabret et al. Efficient matching for content-based publish/subscribe systems. In *Proc. CoopIS*, 2000.
8. Hector Garcia-Molina A.Crespo, and O.Buyukkokten. Efficient Query subscription Processing in a Multicast Environment. In *Proc. ICDE*, 2000.
9. GedML: Genealogical Data in XML. `http://users.iclway.co.uk/mhkay/gedml/`.
10. Himanshu Gupta and Divesh Srivastava. Data Warehouse of Newsgroups. In *Proc. ICDT*, 1999.
11. Eric N. Hanson Chris Carnes, Lan Huang, Mohan Konyala, Lloyd Noronha, Sashi Parthasarathy, J. B. Park, and Albert Vernon. Scalable Trigger Processing. In *Proc. ICDE*, pages 266–275, April 1999.
12. The Intel Corporation. Intel Netstructure XML Accelerators. `http://www.intel.com/netstructure/products/xml_accelerators.htm`, 2000.
13. H.V. Jagadish, Laks V.S. Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A Tree Algebra for XML. *Proc. DBPL*, Roma, Italy, September 2001.
14. Laks V.S. Lakshmanan and P. Sailaja. On Efficient Matching of Streaming XML Documents and Queries. Tech. Report, Univ. Of British Columbia, December 2001. http://www.cs.ubc.ca/ laks/matchmaker-edbt02-full.ps.gz.
15. Laks V.S. Lakshmanan and P. Sailaja. MatchMaker: A system for matching XML documents and queries. Demo paper, *Proc. ICDE*, San Jose, CA, Feb. 2002. To appear.
16. L. Liu C. Pu, and W. Tang. Continual queries for internet-scale event-driven information delivery. *IEEE Trans. on Knowledge and Data Eng.* 11(4): 610-628 (1999).
17. Benjamin Nguyen, Serge Abiteboul, Gregory Cobena, and Mihai Preda. Monitoring XML Data on the Web. *ACM SIGMOD*, 2001.
18. Douglas Terry David Goldberg, David Nichols, and Brian Oke. Continuous queries over Append-only databases. In *ACM SIGMOD*, June 1992.

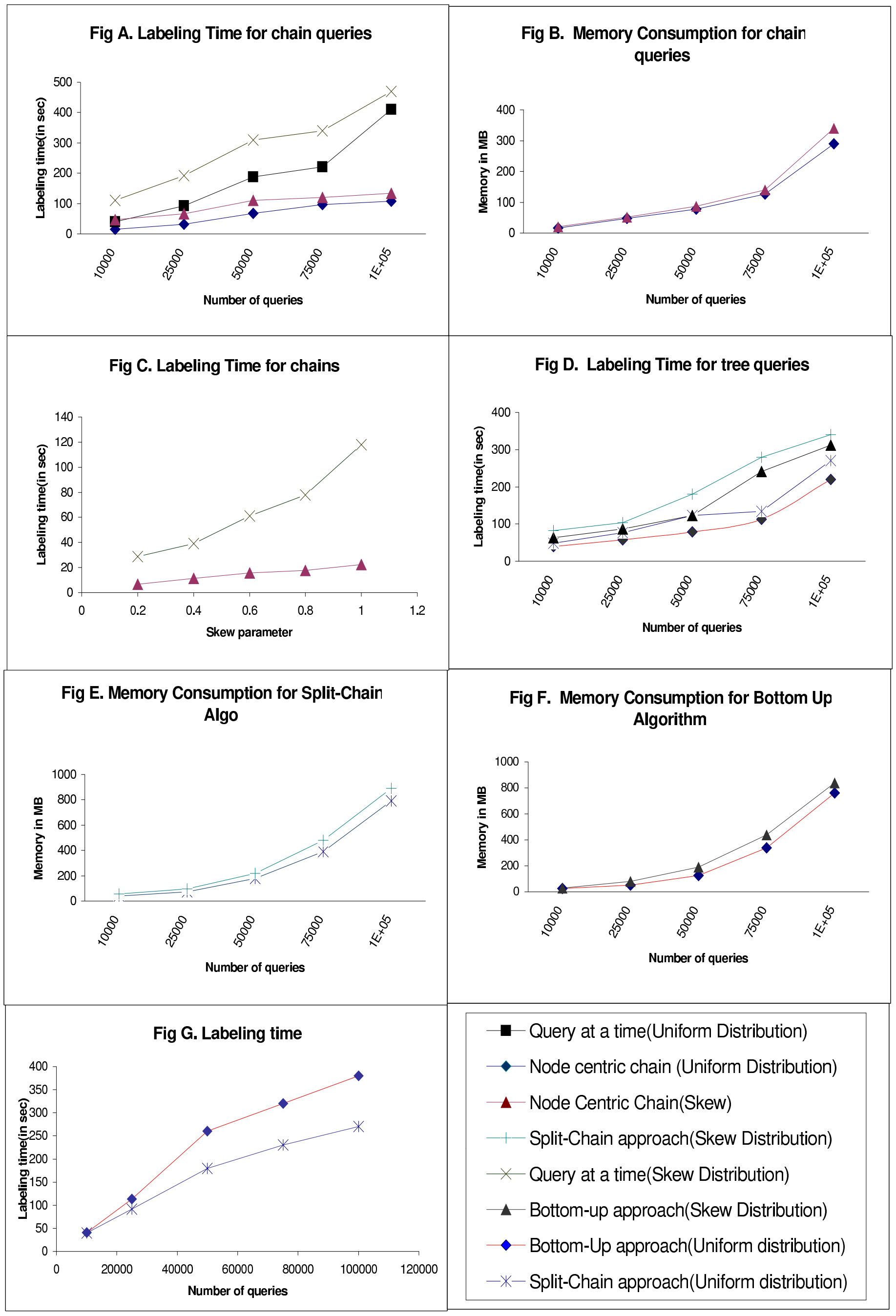

Fig. 7. Experimental Results