SCISSOR: Scalable and Efficient Reachability Query Processing in Time-Evolving Hierarchies

Phani Rohit Mullangi
Department of Computer Science
The University of Georgia
Athens, GA 30602, USA
mullangi@cs.uga.edu

Lakshmish Ramaswamy
Department of Computer Science
The University of Georgia
Athens, GA 30602, USA
laks@cs.uga.edu

ABSTRACT

A time-evolving hierarchy (TEH) consists of multiple snapshots of the hierarchy (collection of one or more trees) as it evolves over time. It is often important to test reachability between a given pair of vertices in an arbitrary (possibly past) snapshot of the hierarchy. While interval-based indexing has been a popular strategy for reachability testing in static hierarchies, a straightforward extension of this strategy to TEHs is impractical because of the exorbitant indexing overheads. In this paper, we propose SCISSOR (***s**ele**c**t**i**ve **s**naps**ho**t indexing with progressive solution **r**efinement*), which, to the best of our knowledge is the first time and space efficient framework for answering reachability queries in TEHs. The main idea here is to maintain indexes only for a selective interspersed subset of TEH snapshots. A query on a non-indexed snapshot will be answered by utilizing the index of a temporally-nearby indexed snapshot and analyzing the structural changes that have occurred between the two snapshots. We also present a experimental study demonstrating the scalability and efficiency of the SCISSOR framework in terms of both indexing costs and query latencies.

Keywords

Time-evolving hierarchies; interval-based indexing; snapshot-specific queries; graph edits

Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

1. INTRODUCTION

Trees and tree collections (henceforth jointly referred to as hierarchies) have long been used for representing hierarchical structures such as XML documents and subclass-superclass relationships in object-oriented software. In many modern applications, however, the hierarchies are not static – they change over time. Consider XML store that manages multiple versions of large XML documents. The hierarchy corresponding to an XML document changes each time a new version is committed to the XML store.

Similarly, a software versioning system will have to deal with class hierarchies that can change from one version to the next especially during the early stages of the software development cycle. Such hierarchies that change over time are called *Time-Evolving Hierarchies* (TEHs). A TEH consists of a sequence of *snapshots* of the hierarchy as it evolves over time.

In these applications, it is often necessary to test reachability (or descendancy) of a given vertex from another vertex on a particular snapshot (which can be *any* snapshot – past or present) of the hierarchy. For example, such a test is necessary for efficiently evaluating an XPath query on a specific version of a multi-version XML document. Similarly, for many software management tasks (e.g., bug tracking) it is necessary to find out whether a given class was a transitive subclass of another class in a particular version of the software. A query that seeks to find out whether a vertex `w` is reachable from another vertex `v` in a given snapshot of a TEH is referred to as a *snapshot-specific reachability query (SS-reachability query).*

The problem of answering reachability queries for static hierarchies has been previously studied in several contexts such as XML query processing [3] and object-oriented (OO) software management. These queries can be answered by an on-demand traversal of the hierarchy (either in breadth-first or depth-first manner) but the query response time is poor for large hierarchies. Researchers have shown that indexing the hierarchies on a relational database can effectively mitigate this problem. Of the several indexing techniques that have been proposed for this purpose [1, 4], interval-based indexing [3] is among the most popular ones.

Shu-Yao Chien et al. [7] have proposed a scheme for managing XML documents that evolve over multiple versions. But to the best of our knowledge they do not address the problem of answering reachability queries. In this paper, we present a *scalable, time-and space-efficient and tunable* indexing framework, called ***s**ele**c**t**i**ve **s**naps**ho**t indexing with progressive solution **r**efinement(SCISSOR)*, for answering reachability queries on any particular snapshot of a TEH. To the best of our knowledge, *SCISSOR is the first framework that can efficiently answer reachability queries on any given snapshot of a TEH.* A key design feature of the SCISSOR framework is that it does not require every snapshot to be indexed.

2. OVERVIEW AND BACKGROUND

As mentioned in the introduction, a TEH consists of a sequence of *snapshots* of a tree collection as it evolves over time. It is assumed that each snapshot of the TEH satisfies the conditions for being a collection of trees (i.e., any vertex in any snapshot has at most one parent). Each edge is assumed to be directed from the parent vertex towards the child vertex. For indexing purposes, tree collections are often converted to a single tree by adding a fictitious super-root that becomes the parent of the roots of individual trees of

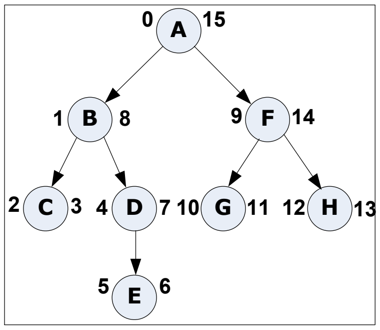

Figure 1: Interval Based Indexing Scheme

the collection. In order to simplify the discussion and without loss of generality, we assume that such a transformation is performed on each snapshot.

Consider a TEH T. Let $\{T_1, T_2, \ldots, T_q, \ldots, T_r\}$ be the different snapshots of the TEH. Let EDList(T_q, T_{q+1}) represent the changes occurring between snapshots T_q and T_{q+1}. Note that the EDList between any two snapshots can be represented as a union of a set of vertex additions, a set of vertex deletions, a set of edge additions and a set of edge deletions, any of which can possibly be empty. We assume that EDList between any two snapshots does not violate the property that each snapshot is a tree collection. In other words, suppose vertex `v` is the child of vertex `u` in snapshot T_q and EDList(T_q, T_{q+1}) includes addition of an edge `(w, v)`, it is assumed that EDList(T_q, T_{q+1}) also includes deletion of the edge `(u,v)`. We also assume that each `EDList` adheres to basic consistency conditions. For example, if an `EDList` contains deletion of a vertex `v`, any incoming and outgoing edges on `v` are also deleted in the same `EDList`.

The SS-reachability query `SSReach(v, w, q)` seeks to find out whether vertex w was reachable from vertex v in the q^{th} snapshot of the TEH. The answer should be TRUE if `w` was reachable from `v` in T_q or FALSE otherwise.

2.1 Interval-based Indexing for Reachability Analysis in Static Trees

There has been considerable interest in efficient answering of reachability queries in static hierarchies. Breadth-first traversal, depth first traversal and transitive closure are among the earliest approaches for answering reachability queries. However, they do not scale well. An alternate approach that has been pursued in recent years is to maintain certain indexing information for the hierarchy on a relational database. Several indexing schemes such as interval-based indexing and HOPI indexing have been proposed for answering reachability queries [3, 4]. Interval-based indexing is one of the most popular schemes because of its simplicity, efficiency and its ability to provide good balance in the trade-off between indexing costs and query-time. Several researchers have proposed schemes that extend interval-based indexing for reachability testing in general directed graphs [5, 6, 8].

Figure 1 illustrates the interval-based indexing scheme. The number to the left of each vertex is its pre-order value whereas the number to right is its post order value. With this index in place, the reachability query `SSReach(v,w)` is true if and only if the pre-order value of `w` is in between the pre and post-order values of `v` ($v\text{pre} < w\text{pre} < v_{\text{post}}$). Notice that $E\text{pre}(= 5)$, $B\text{pre}(= 1)$ and $B_{\text{post}}(= 8)$ satisfies the condition hence E is reachable from B.

2.2 Naive Approaches and their Drawbacks

2.2.1 Index All Snapshot (IAS)

A straightforward approach for answering SS-reachability queries will be to index every snapshot of the TEH using interval-based indexing and check for containment using the condition described in Section 2.1. However, there are many drawbacks to this simple approach. First, the computational overheads of indexing every snapshot is going to be very high, as it will require traversal of each snapshot. Second, the storage overheads are going to be high as well because of the need to save the index values of every snapshot. Both the computational and storage costs are exacerbated as the hierarchies increase in size and as they change more frequently. Third, there may be very few queries on some fraction of the snapshots in which case indexing every version is wasteful both in terms of storage and computation. However, it should also be noted that query distribution is not known apriori. Fourth, with this approach, the applications that use the reachability testing framework have virtually no control over the indexing costs vs. query efficiency tradeoff. In other words, applications cannot *tune* the system to incur less indexing overheads even when they can tolerate small increases in query latencies.

2.2.2 Index No Snapshot (INS)

The other straightforward approach is not maintaining the index corresponding to any snapshot. The queries will be answered by an on-demand traversal of the corresponding snapshots of the hierarchy. The problem with this approach is the very high query latency caused by query-time traversal of the hierarchy.

An ideal approach will not only balance the tradeoff between the indexing costs and query latencies but will also be *tunable* in the sense that the applications should be able to control the tradeoff.

3. REACHABILITY QUERY PROCESSING IN SCISSOR

Without loss of generality, the problem of answering reachability queries in SCISSOR framework is formalized as follows.

Suppose snapshots `q` and `(q+b)` are indexed (i.e., pre- and post-order indexes are available for snapshot `q` and snapshot `(q+b)`). Also suppose the edit list for all intermediate versions between `q` and `(q+b)` are available. The problem now is to answer `SSReach(v, w, q+a)` i.e., whether w was reachable from v in snapshot `q+a` where $0 \leq a \leq b$. If $a = 0$ or if $a = b$, the query is on an indexed snapshot, and it can be answered simply by checking whether $v\text{pre} < w\text{pre} < v_{\text{post}}$ in the corresponding index. If on the other hand, suppose $0 < a < b$. Recall that in this case, our strategy is to first test the reachability between the same pair of vertices on the temporally-closest indexed snapshot (i.e., we modify the query to `SSReach(v, w, q)`) [1], and then to analyze the edits occurring between versions q and (q+a) to check whether the reachability between v and w is impacted by these edits. Through this analysis, we will be able to answer `SSReach(v, w, q+a)`. Throughout this discussion, we use the TEH depicted in Figure 2 as our running example. In this figure, only Snapshot-1 is indexed.

The question is *how do we design a technique for efficiently analyzing the impact of edits occurring between snapshots q and (q+a)?* A trivial way of analyzing the edits is to process *all* edits occurring between versions q and $(q + a)$ in the order they appear in the edit list, and analyze the cumulative effect of these edits on

[1]In this paper, we use the index corresponding to the temporally closest *past* snapshot. Our algorithm can be easily modified to use either temporally-closest past or future indexes.

the reachability of w from v. This approach, however, is not efficient because of two reasons. First, for most queries, it is likely that a large percentage of edits in the edit list are completely unrelated (e.g., occurring at a very different part of the hierarchy), and processing them adds unnecessary overheads. Second, processing an edit requires loading the corresponding part of the tree structure and updating it as per the edit. Thus, processing every edit imposes high memory overheads.

3.1 Observations and Algorithm Overview

We have designed a technique that overcomes these limitations by *analyzing only those edits that are likely to alter the computed value of SSReach(v, w, q) as the hierarchy evolves from version q to version q+a*. The question is how do we correctly figure out which edits impact the reachability of w from v? Edits that seem unrelated at first glance might in fact have an effect on the reachability because of other chronologically subsequent edits. For example in Figure 2, when processing the query SSReach(B, E, 3), the edit `Add(C, G)` might seem unrelated to the query. However, this edit along with `Add(G, E)` alters the reachability from B to E between version 1 and version 3 of the hierarchy.

We make two important observations which will help us accurately identify the edits that are likely to affect the reachability value from v to w as the graph evolves from snapshot q to $(q + a)$.

- First, suppose `SSReach(v, w, q)` is TRUE. This reachability status can be changed (that w is made unreachable from v) only by deletions of edges incident upon w or w's ancestors until the vertex v.
- If suppose `SSReach(v, w, q)` evaluates to be FALSE, any set of edits occurring between versions q and q+a that can alter the reachability status (i.e, make w reachable from v) will contain at least an edit that adds an inward edge to w or one of its ancestors until (and including) the root of w's current tree.

In fact these two observations are applicable even at intermediate stages (after subset of edits has been processed). We incorporate these observations into our algorithm through a unique concept called *impact list*. The impact list for node pair (v, w) (represented as `ImList(v, w)`) is a dynamically changing list of vertices with the following important property. *At any point of the algorithm, if an arbitrary vertex u does not appear in the ImList(v, w) then it is guaranteed that edits involving vertex u will not affect the reachability status in the algorithm.* `ImList(v, w)` is dynamic in the sense that it is updated each time an edit is processed. However, the ImList satisfies the following two invariants at each stage of the algorithm. If w is currently reachable from v, ImList contains w and all ancestors of w until vertex v. If w is not currently reachable from v, ImList contain all ancestors of w until the root of w's tree. Notice that these two invariants are direct implications of the two observations we stated earlier. ImList is used repeatedly in our algorithm to dynamically select the next edit from the edit list for processing. ImList(v, w) is constructed using the available index for version q based on the results of SSReach(v, w, q), and it is updated each time an edit is processed. (as explained in the algorithm description below). Consider there are p edits between version q and q+a. In the worst case scenario all p edits needs to be processed in order to answer the query hence complexity of answering the queries grows linearly with number of edits.

3.2 Reachability Testing Algorithm

We now outline our algorithm for efficiently processing the edits. First, we explain three important notations that we will use

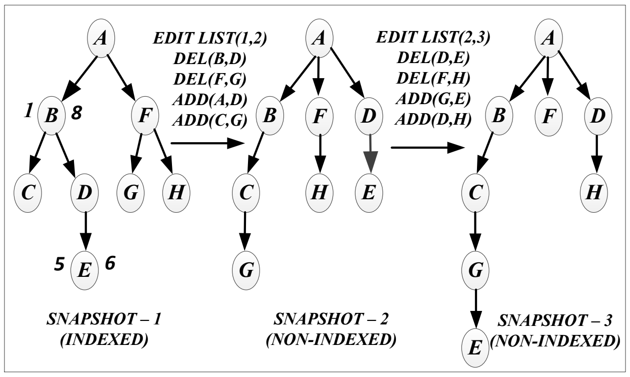

Figure 2: Illustration of SCISSOR framework

in our algorithm description. The cumulative edit list for version q+a (denoted as `CEDList(q+a)`) contains all the edits occurring between the snapshot q (temporally closest indexed snapshot) and snapshot (q+a). The `Reachability_Status` a variable that holds the reachability status between v and w as the algorithm progressively processes various edits. The current ancestor list of an arbitrary vertex u at any point of the algorithm (represented as `CANList(u)`) denotes the list of u's ancestors in the increasing order of their distances to u at that point of the algorithm (i.e., taking into account the effect of edits that have already been processed). Note that, in our scheme, `CANLists` for most vertices need not be explicitly stored – they can be computed by using the index for snapshot q. `CANList` of a vertex has to be stored explicitly only if its ancestors have changed as a result of the edits that have been processed thus far.

Algorithm 1 shows the pseudo-code of our algorithm. In the initialization phase of the algorithm(lines 1-2), we first compute the `CEDList(q+a)` by concatenating the edit lists of snapshots (q+1) through (q+a). `CEDList(q+a)` is also pre-processed to eliminate any pairs of edits that negate each other. In lines 3-11, we handle the special case when the query's destination node w is not existing in snapshot q. This node might have been added between snapshots q and (q+a) (i.e., the `CEDList(q+a)` will include `Add(w)`) or w might still be non-existent in snapshot (q+a) (i.e., the `CEDList(q+a)` will not include `Add(w)`). If w was added between snapshots q and (q+a), (lines 4-7) `Reachabilit y_Status` is set to FALSE, `ImList(v, w)` is initialized to include just w and the algorithm proceeds to the iterative phase. On the other hand, if `CEDList(q+a)` does not include `Add(w)`, the algorithm terminates immediately with `Reachability_Statu s` set to FALSE (lines 7-9). This is because the destination node of the query is not even present in the snapshot (q+a). In lines 12-19, we handle the case when w exists in snapshot q. In this case, we evaluate `SSReach(v, w, q)` and assign it to `Reachability _Status`. If `SSReach(v, w, q)` is TRUE then `ImList(v, w)` is initialized to w followed by the ancestors of w leading up to v in the increasing order of their distance from w(i.e., w is followed by its immediate parent and so on until v). If `SSReach(v, w, q)` is FALSE, `ImList(v, w)` is initialized to w followed by all the ancestors of w in the increasing order of their distance from w.

The algorithm then enters the iterative phase (lines 20-42) wherein the edits in the `CEDList(q+a)` are processed. The choice of the edits that are to be processed in any iteration is determined by the value of `Reachability_Status` and the current composition of `ImList(v, w)`. Suppose `Reachability_Status` is TRUE at the beginning of the ith iteration the algorithm per-

Algorithm 1 SSReach($v, w, q + a$)

```
1:  $CEDList \leftarrow Initialize\ with\ edits\ between\ q\ and\ q + a$
2:  $CEDList \leftarrow Pre - process\ (CEDList)$
3:  if $w$ not in snapshot $q$ then
4:      if $CEDList$ contains $Add(w)$ then
5:          $ImList \leftarrow w$
6:          $Reachability\_Status \leftarrow false$
7:      else
8:          $Reachability\_Status \leftarrow false$
9:          exit
10:     end if
11: else
12:     if $v^q_{pre} < w^q_{pre} < v^q_{post}$ then
13:         $Reachability\_Status \leftarrow true$
14:         $ImList \leftarrow fetchAncestors(v, w)$
15:     else
16:         $Reachability\_Status \leftarrow false$
17:         $ImList \leftarrow fetchAncestors(w)$
18:     end if
19: end if
20: while $i <$ Size of $ImList$ do
21:     $vertex \leftarrow ImList[i]$
22:     if $Reachability\_Status == TRUE$ then
23:         $delete\_edit \leftarrow SearchCEDList(vertex)$
24:         if $delete\_edit$ is not $null$ then
25:             $Imlist \leftarrow process(delete\_edit)$
26:             if $Imlist$ doesn't contain $v$ then
27:                 $Reachability\_Status \leftarrow FALSE$
28:             end if
29:             $CEDList \leftarrow remove(CEDList, delete\_edit)$
30:         end if
31:     else
32:         $add\_edit \leftarrow SearchCEDList(vertex)$
33:         if $add\_edit$ is not $null$ then
34:             $Imlist \leftarrow process(add\_edit)$
35:             if $Imlist$ contains $v$ then
36:                 $Reachability\_Status \leftarrow TRUE$
37:             end if
38:             $CEDList \leftarrow remove(CEDList, add\_edit)$
39:         end if
40:     end if
41:     $i++$
42: end while
```

forms the following actions (lines 22-30). For the vertex being considered, we check the `CEDList(q+a)` to see if there is an `Delete` edit with the vertex under consideration as the destination of the edge being deleted. Suppose x is the first vertex on the `ImList(v,w)` that has a `Delete` edit. We process this edit by performing the following actions. First, all the vertices beyond `x` (not including `x`) are removed from `ImList(v, w)`. Second, for each descendant of the vertex `x`, we materialize its `CANList` if not already materialized. We also update the `CANList` of each descendant of `x` to reflect the edit (i.e., we remove vertices beyond the vertex `x` from each `CANList`). Finally, `Reachability_Status` is set to FALSE and the edit itself is removed from `CEDList(q+a)`.

If `Reachability_Status` is FALSE at the beginning of the i^{th} iteration, line 31 through 40 in the pseudo-code are executed. `ImList` is again scanned from left to right (i.e., considering vertices in the increasing order of their distances to w). But now, for each vertex being considered, we check `CEDList(q+a)` to see if there is an an *Add* edit with the vertex under consideration as the destination of the edge being added. Suppose y is the first such vertex on the `ImList`. Let the added edge be (z, y). In order to process this edit, we first check whether y has a parent in the `ImList` (i.e., y is not the root). If so, there must be an unprocessed remove edit with y as the destination (since every version is assumed to be a collection of trees). We find that edit, remove it from `CEDList(q+a)` and remove all the vertices to the right of y from the current `ImList`. Next, we check whether z is currently reachable from v (this can be done by checking `CANList(z)`, if materialized or through containment checks on snapshot q). If z is not reachable from v, we append z and all the parents of z to the `ImList` in the increasing order of their distance to z. Since z itself is not reachable from v, the `Reachability_Status` remains as FALSE. If, on the other hand, z is reachable from v, `Reachability_Status` is set to TRUE. We also append z and all the parents of z to vertex v to the `ImList` in the increasing order of their distance to z. The CANLists of all descendants of y are materialized and updated to reflect the new ancestors of y.

As stated above our algorithm terminates under two distinct conditions. They are

1. `Reachability_Status` is FALSE and there are no `Add` edits involving the vertices in `ImList`; or

2. `Reachability_Status` is TRUE and there are no `Delete` edits involving the vertices in `ImList`.

Upon termination, the value of the `Reachability_Status` holds the result of `SSReach(v, w, q+a)`. Note that the value of `Reachability_Status` may flip multiple times during the algorithm.

We now illustrate our algorithm on the TEH shown in Figure 2 as it evolves through snapshots 1 through 3 (only snapshot-1 is indexed). Suppose we need to answer the query `SSReach(B,E,3)`. Snapshot-1 is the nearest indexed snapshot, and hence it is used as the initialization point. `Reachability_Status` is set to TRUE as $B_{\text{pre}} < E_{\text{pre}} < B_{\text{post}}$ in the index corresponding to Snapshot-1. CEDList(3) is initialized to `{DEL(B,D),DEL(F,G),DEL(D,E),DEL(F,H),ADD(A,D),ADD(C,G),ADD(G,E),ADD(D,H)}` and ImList is initialized to `{E,D}`. Since `Reachability_Status` is TRUE at the beginning of the first iteration, the algorithm scans the ImList from left to right to check if an inward edge on one of the vertices is removed by a DELETE entry in CEDList(3). E is the leftmost vertex to have a DELETE edit (`DEL(D,E)` is the corresponding edit). As a result of processing this edit, `Reachability_Status` becomes FALSE, ImList is truncated to `{E}` and `DEL(D,E)` is removed from CEDList(3). In the second iteration, the algorithm scans the ImList from left to right looking for a vertex for which an edit on CEDList(3) adds an inbound edge. The algorithm processes the edit `ADD(G,E)` by expanding the ImList to `{E,G,F,A}` and deleting the entry from CEDList(3). Note however that `Reachability_Status` still remains FALSE because G is not reachable from B as yet (the edit ADD(C, G) is not yet processed). Thus, in iteration 3 we continue to look for more vertices on the modified ImList with corresponding ADD entries on CEDList(3). The edit `ADD(C,G)` is now found. However, since G already has a parent vertex, namely F, the entry `DEL(F,G)` should exist in CEDList(3) so as to maintain the "tree" property. This edit is found and processed along `ADD(C,G)`. As a result the ImList is modified to `{E, G, C}` and `Reachability_Status` is set to TRUE (because C is reachable from B). The entries `ADD(C,G)` and `DEL(F,G)` are both removed from CEDList(3). The algorithm enters the fourth iteration with `Reach`

Size	Index No Snapshot (INS)		Index All SnapShots (IAS)		SCISSOR (1/100)	
	Indexing Time (sec)	Querying Time (msec)	Indexing Time (sec)	Querying Time (msec)	Indexing Time (sec)	Querying Time (msec)
10K	0	12.73	1.33	2.28	0.02	7.86
100K	0	156.70	12.66	10.16	0.13	18.01
250K	0	628.29	31.53	19.74	0.33	28.41
500K	0	1796.15	62.90	35.26	0.67	44.74

Table 1: Indexing and Querying latencies of INS, IAS and SCISSOR

ability_Status = TRUE, ImList = {E,G,C} and CEDList(3) = {DEL(B,D),DEL(F,H),ADD(A,D),ADD(D,H)}. In this iteration, the algorithm scans the ImList from left to right looking for a vertex for which there is a corresponding DELETE entry on CEDList(3) that removes an inbound edge. Since there are no such entries currently on CEDList(3), the algorithm terminates with the answer to the query SSReach(B,E,3) being TRUE (the last value carried by Reachability_Status). Note that the algorithm processed only 4 out of the 8 edits in the original CEDList(3) (as initialized at the beginning of the algorithm). This illustrates that our algorithm does not process edits that are clearly irrelevant to the query thereby avoiding unnecessary overheads.

4. EXPERIMENTS AND RESULTS

We now discuss the experiments we have performed to evaluate the SCISSOR framework. We have implemented the SCISSOR framework in Java. The implementation is done in a modular fashion so that specific components can be enabled or disabled for evaluation. We compare SCISSOR with two other approaches discussed in Section 2, namely, index all snapshots (IAS) and index no snapshot (INS). IAS can be realized within the SCISSOR framework by indexing every snapshot of a TEH. We have implemented INS in two ways – one uses breadth-first traversal (BFT) on the queried snapshot while the other uses depth-first traversal (DFT). Both these implementations were in Java and they yielded very similar results for most experiments. Unless otherwise mentioned, the results reported for INS correspond to the DFT-based implementation.

To the best of our knowledge, there are no publicly available TEH data sets. Thus, we have had to rely upon synthetic data sets for our experimental evaluation. However, in order to comprehensively study the behavior of the proposed framework, we generate a number of data sets by varying important parameters of TEHs (e.g., hierarchy size, height, total number of snapshots etc.). The generator program accepts hierarchy size (in terms of number of vertices), hierarchy height, number of snapshots, and average number of edits between snapshots as input parameters. Based on the specified hierarchy size and hierarchy height, we generate a tree with each non-leaf vertex having approximately same number of children and designate that as the first snapshot. We create subsequent snapshots by generating certain number of edits (as determined by the corresponding parameters) on the previous snapshot. While creating snapshots, we ensure that each of them is a tree or collection of trees. Each edit is generated as follows. First, we need to decide the type of edit. It can be an *edge-add* edit or *edge-delete* edit. We select the type of edit based on the desired ratio of *edge-add* and *edge-delete* edits per snapshot. To generate an add edit, two vertices are chosen randomly and an edge is added from the first vertex to the second if an edge doesn't already exist between them and remove any inbound edge on the second vertex. To generate an delete edit, an edge is randomly chosen from set of existing edges and deleted. Edit lists used in our experiments have approximately the same percentage of add and delete edits.

The query workload is generated in the following manner. For each query, the source vertex, the destination vertex and the snapshot are all chosen randomly. However, random selection results in a large fraction of "unreachable" queries (queries with negative answer). The workload is adjusted (by dropping some randomly chosen unreachable queries) to have approximately equal percentages of reachable and unreachable queries.

We evaluate SCISSOR on two main performance aspects, namely indexing overhead and query latency. We use *amortized indexing latency* as the metric for quantifying indexing overhead. Suppose a TEH has n snapshots of which k are indexed and the total time taken for indexing all k of them is T time units. Amortized indexing cost for this scenario is $\frac{T}{n}$. In order to provide better insight into query processing overheads in SCISSOR, we measure the latency incurred from database querying and the latency incurred by edit list processing. In each case, we report the mean over all queries. We also measure the fraction of edits that our algorithm processes while answering a particular query to validate our claim that SCISSOR processes only a fraction of edits that have occurred between the queried and the nearest-indexed snapshots. *Amortized storage cost* is the ratio of the total disk-space needed to store indexes and edit lists over all snapshots to the number of snapshots of the TEH (n).

Benefits of Selective Indexing: In this set of experiments, we quantify the benefits of the SCISSOR framework when the number of vertices in the TEH varies from 10K to 500K. The height of all hierarchies is set to 10. The total number of snapshots for each TEH in this experiment is 100. For each TEH, we compare the performances of the SCISSOR framework, the IAS scheme and the INS scheme. For the SCISSOR framework, we vary the fraction of snapshots that are indexed and measure the performance of system by executing 10000 queries which are randomly distributed across all the snapshots.

Table 1 represents indexing and querying latencies of INS, IAS and SCISSOR schemes. INS has the highest query latency, while IAS has the highest indexing latency. SCISSOR (indexing every 100^{th} version), on the other hand, yields substantial savings in indexing costs ($> 90\%$ for a 500K node TEH) with marginal increase ($< 12\%$ for the same TEH) in query latencies when compared with the other approaches. In the following experiments we show how SCISSOR balances the query and indexing latencies.

Figure 3 indicates the amortized indexing costs of SCISSOR and IAS schemes varying vertices from 10K to 500K vertices. The X-axis indicates, log-scale of the percentage of snapshots indexed in the SCISSOR framework. As we have mentioned before, when the fraction of indexed snapshots is set to 1, the SCISSOR framework behaves exactly like the IAS scheme. As expected, IAS has the highest amortized indexing cost (around 62 secs per snapshot). For the SCISSOR framework, amortized indexing costs fall almost lin-

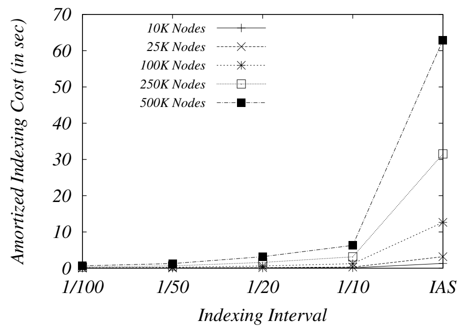

Figure 3: Amortized Indexing Cost (varying vertices)

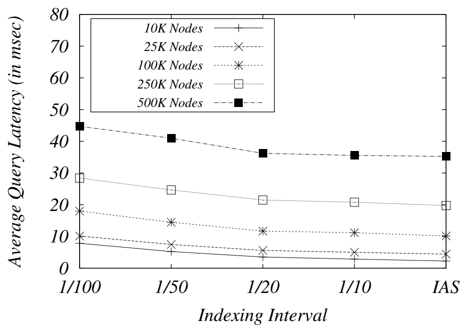

Figure 4: Avg Query Latency (varying vertices)

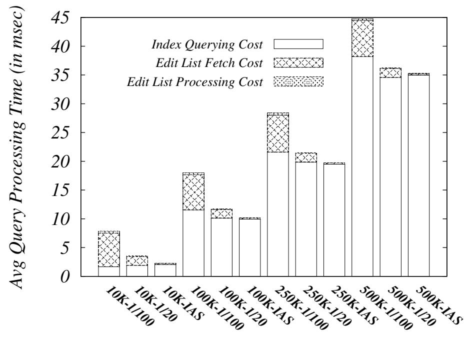

Figure 5: Split Query Processing Time (varying vertices)

early with fraction of indexed snapshots. This behavior is entirely due to the number of snapshots that need to be indexed. The indexing costs of INS on the other hand are zero(not shown in figure), because it does not index any snapshot.

Figure 4 shows the average query latencies from the same experiment. The results of IAS and SCISSOR, on the other hand, may seem intuitive. As expectecd IAS has the least query latency values because it can obtain the answer just by issuing an appropriate DB query as all the versions are indexed and SCISSOR's latency values shows a marginal increase because of the edits that needs to be processed. In order to further clarify the point regarding query latencies, stacked histogram in Figure 5 represents three types of latencies, first is the latency involved in querying the nearest indexed snapshot followed by the latency involved in fetching the edits between the nearest indexed snapshot and the queried snapshot and the latency involved in processing the edits. A Data point (10K-1/100) on X-axis indicates that size(10K) of the dataset followed by fraction of snapshots indexed (every 100^{th} snapshot). The noteworthy points in Figure 5 are (1) edit list processing latency is a very minor contributor to the average query latency (2) edit list processing latency increases as the fraction of indexed snapshots is reduced.

Since SCISSOR indexes only a subset of snapshots, it also provides significant disk space savings. Amortized storage costs are the highest for IAS as it needs to store indexes for all snapshots. For SCISSOR, on the other hand, our experiments indicate that the amortized storage costs vary almost linearly with indexing frequency.

5. RELATED WORK

Efficient processing of reachability queries for static trees and more generally, for static graphs has been an active topic of research. GRIPP [5], DualLabeling [6] and GRAIL [8] are the recent interval-based approaches for answering reachability queries in graphs. Each of them augment the basic interval-based approach in a different manner to account for additional connectivity provided by non-tree edges. DualLabeling augments interval-based indexing with a transitive link table (TLT), while GRIPP uses a reachability instance set to recursively traverse paths composed of non-tree edges. GRAIL on the other hand maintains indexes corresponding to multiple spanning trees. While Schekel et al. discuss incremental maintenance of the HOPI index [4], it can only be used to answer queries on the current DAG and not on previous snapshots. SCISSOR, on the other hand, can answer reachability queries on any snapshot.

6. CONCLUSION

Efficient and scalable processing of reachability queries in time-evolving hierarchies is important for many modern applications. In this paper, we presented a tunable, time and space efficient framework called SCISSOR for testing reachability between given pair of vertices on any given snapshot of a TEH. The main idea behind SCISSOR is to selectively index a subset of TEH snapshots and use these snapshot indexes to answer reachability queries on all snapshots of the TEH.

Acknowledgment

This research has been partially funded by the National Science Foundation under Grant Number CNS-1338276. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors, and do not necessarily reflect the views of the NSF.

7. REFERENCES

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD Conference*, 1989.

[2] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5), 2003.

[3] T. Grust, M. van Keulen, and J. Teubner. Accelerating xpath evaluation in any rdbms. *ACM Trans. Database Syst.*, 29, 2004.

[4] R. Schenkel, A. Theobald, and G. Weikum. Hopi: An efficient connection index for complex xml document collections. In *EDBT*, 2004.

[5] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD Conference*, 2007.

[6] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual Labeling: Answering Graph Reachability Queries in Constant Time. In *ICDE*, 2006.

[7] S. yao Chien and V. J. Tsotras. Storing and querying multiversion xml documents using durable node numbers. In *Proc. of WISE*, pages 270–279. WISE, 2001.

[8] H. Yildirim, V. Chaoji, and M. J. Zaki. GRAIL: Scalable Reachability Index for Large Graphs. *PVLDB*, 3(1), 2010.