Efficient Reachability Analysis for Evolving Graph Structured Data

by

Phani Rohit Mullangi

(Under the direction of Prof. Lakshmish Ramaswamy)

Abstract

Reachability analysis is a fundamental operation in many applications that work with graphs as underlying data structure. It has been extensively studied from the context of static graphs. In most modern computing domains, these graphs are no longer static (they evolve over time) and they are massive. Existing platforms and algorithms built for answering reachability queries in static graphs results in huge performance costs when applied on evolving graphs. The large size of graphs and their evolving nature calls for new, scalable and efficient algorithms/systems for answering reachability queries. In this research, we propose a generic, scalable, time-and space-efficient frameworks for reachability analysis in massively evolving graph structured data. We leverage our framework for answering queries in large evolving XML documents.

We have created frameworks that can efficiently answer snapshot-specific reachability queries as well as continuous reachability queries in evolving graphs, where a snapshot-specific reachability query seeks to find out whether a vertex w is reachable from another vertex v in a given snapshot of a structure and a continuous reachability queries are issued only once and then logically run continuously over the evolving data set to find the status(reachable/unreachable) of given set of queries.

We introduce SCISSOR as a generic indexing and querying framework for answering snapshot-specific queries in large time-evolving hierarchies. The central idea is to maintain indexes for an interspersed subset of snapshots. An reachability query on a non-indexed snapshot will be processed by first answering the same query on the temporally-closest indexed-snapshot, and then progressively modifying the solution to reflect the effects of the changes occurring between the queried snapshot and indexed snapshot. We have also designed a highly efficient, scalable and provably correct algorithm for analyzing the effects of changes between the queried and the nearest indexed snapshot. we leverage this framework to propose a tunable, time and space efficient algorithms to evaluate XPath expressions on continuously evolving XML document (CEXML) repositories. We introduce a new class of XPath expression queries for CEXML document called version-specific XPath queries.

We introduce CoUPE as a generic indexing and querying engine for answering continuous queries in evolving graphs. We have designed an efficient algorithm for updating the indices of graph by analyzing the changes happening to the graph. These indices combined with a novel heuristic helps us figuring out the status of the queries.

INDEX WORDS: Multi-version XML, XPath Expressions, Interval based indexing

Efficient Reachability Analysis for Evolving Graph Structured Data

by

Phani Rohit Mullangi

A Dissertation Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Doctor of Philosophy

Athens, Georgia

2015

EFFICIENT REACHABILITY ANALYSIS FOR EVOLVING GRAPH STRUCTURED DATA

by

PHANI ROHIT MULLANGI

Approved:

Major Professors: Lakshmish Ramaswamy

Committee: Liming Cai
Kang Li

Electronic Version Approved:

Julie Coffield
Interim Dean of the Graduate School
The University of Georgia
May 2015

Efficient Reachability Analysis for Evolving Graph Structured Data

Phani Rohit Mullangi

April 26, 2015

Acknowledgments

I would like to thank my advisor, Prof. Lakshmish Ramaswamy, for crucial research suggestions and guidance. Also, I would like to thank professors in my advising committee Prof. Liming Cai and Prof. Kang Li for their support and valuable suggestions during the years I was trying to make progress in my research. I am also grateful to all the faculty and staff members of the computer science department.

Furthermore, I appreciate support and patience of my wife during my PhD program. I am also grateful to my parents for all their help and support not only during this adventure, but through out my whole life. Also, I would like to thank my cousins (Ram and Krishna) for their incredible support through out my PhD program. Cheers to my friends for always being there for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graph is a fundamental data structure for representing information in many domains. Social networks communication networks, the World Wide Web and Life Sciences are some of the popular domains which consists of massively large graphs. The importance of querying massively large graphs for understanding and extracting useful pieces of information has evoked a huge interest in the recent past among many research communities. Querying such big graphs for extracting useful insights need highly scalable and efficient algorithms.

Reachability(or descendency) query processing forms an important class of graph queries across all the domains. In simple words, a query that seeks to find out whether a vertex `w` is reachable from another vertex `v` in the graph is called a reachability query. Software build dependency management systems, XML repositories, Maps etc are the few of the applications that extensively use reachability queries.

In most modern computing applications, these graphs/hierarchies are not static they evolve over time. For example, consider a social network graph that changes every time when new users are added to the network or new users are added to an existing user's network. Such a graph is called *Time-Evolving Graph*(TEG). Also, consider a 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 software development cycle. Such hierarchies that change over time are called *Time-Evolving Hierarchies* (TEHs, for short). Similarly A TEH/TEG consists of a sequence of *snapshots* of the graph as it evolves over time. We refer to such structures as evolving graph structured data.

In all these applications, it is often necessary to test reachability of a given vertex from another vertex on a particular snapshot (which can be *any* snapshot – past or present) of the evolving structure. We refer to such queries as reachability queries on evolving graph structured data. There are different types of reachability queries that can be asked on an evolving graphs as mentioned below:

- A query that seeks to find out whether a vertex `w` is reachable from another vertex `v` in a given snapshot of a structure is referred to as a *snapshot-specific reachability query*. These queries are run only once to completion.

- Queries are issued only once and then logically run continuously over the evolving data set are called *continuous queries*. The initial state of reachability query issued at time instance T_1 is evaluated based on the state of the graph at same time instance(T_1). Status of these queries needs to be reevaluated as the underlying graph changes.

The problem of answering reachability queries for static graphs has been previously studied in several contexts such as XML query processing [35] and object-oriented (OO) software management [5]. These queries can be answered by an on-demand traversal of these structures (either in breadth-first or depth-first manner) but the query response time is poor for large data sets (breadth-first and depth-first traversals are both $O(N)$ where N is the number of vertices in the hierarchy). Researchers have shown that indexing the graphs on a relational database can effectively mitigate this problem. Of the several indexing techniques that have been proposed for this

purpose [3, 35, 64], interval-based indexing [35] (a.k.a pre- and post-order indexing) is among the most popular ones. But these existing approaches used for computing reachability queries on static structures doesnt scale well directly in the context of evolving graph structured data. In order to overcome these problems we propose framework for efficiently answering reachability queries.

First, we present a *scalable, time-and space-efficient and tunable* indexing framework, called ***s**elec**t**ive **s**naps**h**o**t** indexing with progressive solution **r**efinement(SCISSOR)*, for answering reachability queries on any particular snapshot of a Time Evolving Hierarchy (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 of evolving graph to be indexed.Hence SCISSOR can be used as a generic indexing and querying framework for answering snapshot-specific queries in large time-evolving hierarchies. We leverage our generic SCISSOR framework to propose a tunable, time and space efficient algorithms to evaluate XPath expressions on continuously evolving XML document (CEXML) repositories. We introduce a new class of XPath expression queries for CEXML document called *version-specific* XPath queries and propose a set of scalable and efficient algorithms for answering most important types of version-specific XPath queries, namely, *ancestor, descendent, preceding* and *following*, on any given version of a continuously evolving XML (CEXML) document.

Finally, we present a processing engine called (CoUPE) for answering continuous reachability queries where continuous queries are issued only once and then logically run continuously over the evolving data set. CoUPE consists of highly efficient and scalable algorithm for updating the state of connectivity (indices) of evolving graph and a novel heuristic for answering continuous reachability queries.

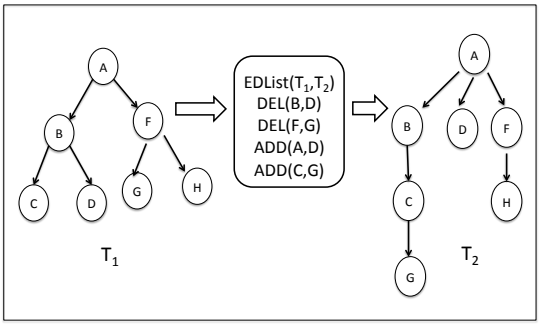

Figure 1.1: Illustration of Time Evolving Hierarchy (TEH)

1.1 Overview and Background

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.

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`. Figure 1.1 illustrates the TEH. *DEL* represents an edge deletion and *ADD* represents an edge addition.

Chapter 2

Scalable and Efficient Reachability Query Processing in Time-Evolving Hierarchies

2.1 Introduction

As mentioned earlier, TEH consists of a sequence of *snapshots* of a tree collection as it evolves over time. In this chapter, we present a *scalable, time-and space-efficient and tunable* indexing framework, called ***s**elec**t**ive **s**naps**h**o**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. The contributions of this work are fourfold.

- First, we introduce SCISSOR as a generic indexing and querying framework for answering snapshot-specific queries in large time-evolving hierarchies. The central idea is to maintain indexes for an interspersed subset of snapshots. An reachability query on a non-indexed snapshot will be processed by first answering the same query on the temporally-closest indexed-snapshot, and then progressively modifying the solution to reflect the effects of

the changes occurring between the two snapshots.

- Second, we outline two important observations regarding the characteristics of structural changes of a hierarchy and their potential impact on reachability queries. These observations help us to significantly narrow down the changes that need to be processed when answering reachability queries thereby laying the foundation for a highly efficient reachability testing algorithm.

- Third, we design a highly efficient, scalable and provably correct algorithm for analyzing the effects of changes between the queried and the nearest indexed snapshot on the result of the reachability query. Our algorithm is based on a unique concept called *impact list* which is a dynamic list of vertices which guides us in selecting the next change to be processed at each stage of the algorithm. Furthermore, the impact list also helps us determine when the algorithm can terminate with a final answer.

- Fourth, we present a novel heuristic-based technique for deciding which snapshots to index. At the core of our technique is a unique metric for quantifying the percentage of unique reachability queries that can potentially be affected by an individual change in the structure of the hierarchy.

We have performed a range of experiments to comprehensively study the performance of the SCISSOR framework. The results of the experiments show that SCISSOR 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 approach of indexing every snapshot of the TEH.

2.2 Overview and Background

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

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 (i.e., whether `w` was a descendant of `v` in the q^{th} snapshot). The answer should be TRUE if `w` was reachable from `v` in T_q or FALSE otherwise.

2.2.1 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 [35, 64]. 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 [20, 40, 71, 75, 80].

Figure 4.1 illustrates the interval-based indexing scheme. The number to the left of each vertex is its pre-order value whereas the number to its 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` (i.e., $v_{\text{pre}} < w_{\text{pre}} < v_{\text{post}}$). Notice that $E_{\text{pre}}(= 5)$, $B_{\text{pre}}(= 1)$ and $B_{\text{post}}(= 8)$ satisfies the condition hence E is reachable from B.

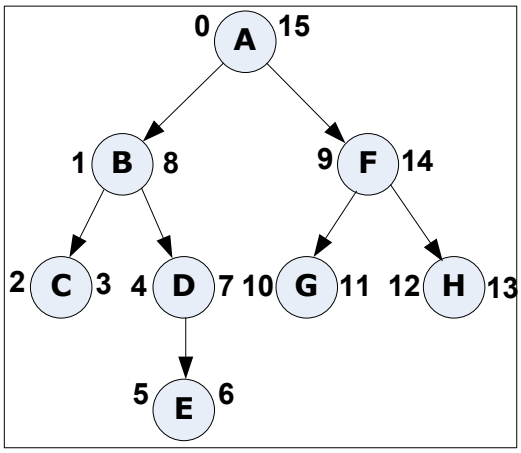

Figure 2.1: Interval Based Indexing Scheme

2.2.2 Naive Approaches and their Drawbacks

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 3.1.3. 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 certain 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 in terms of snapshots 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.

Index No Snapshot (INS) : The other straightforward approach is not maintaining 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.

2.3 SCISSOR Technique

In this section we discuss our framework for answering reachability queries in time-evolving hierarchies called ***s**elec**t**ive **s**naps**h**ot indexing with progressive solution **r**efinement (SCISSOR)*. As the name suggests, the main idea here is to selectively index interspersed snapshots of the TEH. These subsets of snapshots are henceforth referred to as indexed-snapshots. These indexes together with the list of changes occurring between snapshots will be used for answering reachability queries on any snapshot of the TEH. Two important questions need to be addressed when designing the SCISSOR framework – ***(1)** How to answer SS-reachability queries on a particular snapshot, especially if the snapshot is not indexed?; and **(2)** How to decide which snapshots should be selected for indexing?*. Before we address these questions, we give a high-level overview of the SCISSOR framework.

Figure 3.3 shows the high-level architecture of the SCISSOR framework. The applications using the SCISSOR framework provide the complete hierarchy (vertices and edges) for the first snapshot. For each subsequent new snapshot, the application just provides the changes between the previous snapshot and the current snapshot in the form of an edit list. The edit list between snapshot q and snapshot (q+1) is represented as EDList$(q, q+1)$ and it contains 4 different types of entries `Add(u v)` indicating the addition of an edge from u to v, `Delete(u v)` indicating the removal

of the edge from u to v, `Add(u)` indicating the addition of node u and `Delete(u)` indicating the deletion of node u. It is assumed that all edit lists adhere to basic consistency requirements. For example, if EDList$(q, q+1)$ contains `Delete(u)`, it will also have deletions of any incoming and outgoing edges on node u. We also assume that no edit list violates the hierarchy constraints [1]. The application controls the indexing costs vs. query efficiency tradeoff through the parameter ρ. The higher the value of ρ the lower the number of snapshots that get indexed, and vice-versa.

The SCISSOR framework has three major components as shown in Figure 3.3. The first is the SS-reachability query processing component which uses the available indexes and edit lists to answer incoming queries. The second component of our framework is an online heuristic-based technique to decide whether to index an incoming new snapshot. If the decision is not to index the incoming snapshot, the edit list corresponding to this snapshot is stored in the DB. If on the other hand, the decision is to index the incoming snapshot, the edit list is passed to the third component called indexing component. The indexing component generates new pre- and post-order index values for the incoming snapshot and stores the new index values on the DB.

2.4 Reachability Query Processing in SCISSOR

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

Figure 3.2 represents the timeline of indexed snapshots of a evolving hierarchy. Snapshots `q`, `(q+b)`, `(q+c)`... are indexed (i.e., pre- and post-order indexes are available for snapshots `q`, `(q+b)`...). Also the edit list for all intermediate versions between `q`, `(q+b)` etc. 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 ≤ a ≤ b.

If $a = 0$ or if $a = b$, the query is on an indexed snapshot, and it can be answered simply by

[1] Violations of these assumptions can be easily checked.

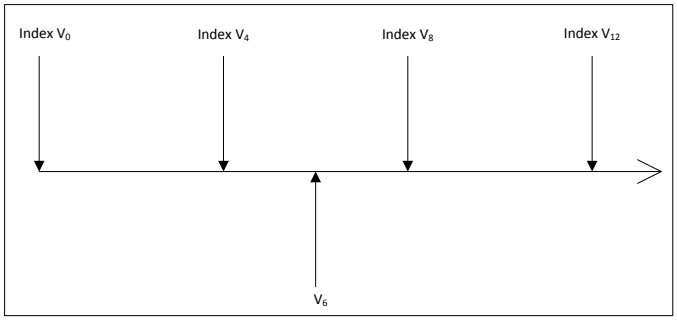

Figure 2.2: Timeline of snapshots

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)`) [2], 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 2.4 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 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.

[2] Throughout this chapter, 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.

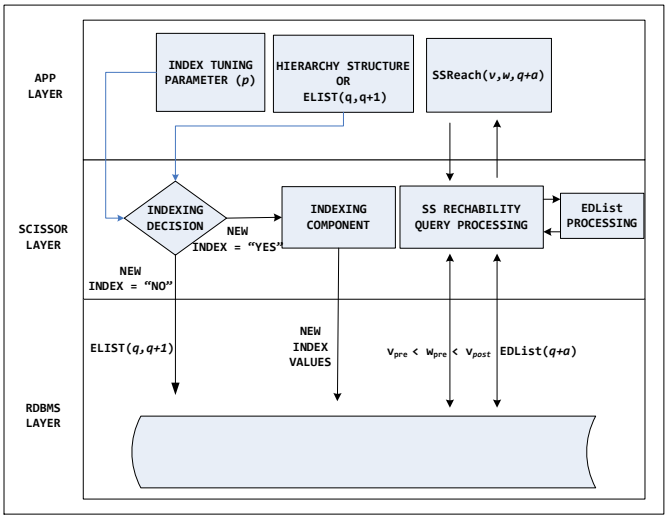

Figure 2.3: Architecture of the SCISSOR framework

2.4.1 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 2.4, 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 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.

2.4.2 Reachability Testing Algorithm

We now outline our algorithm for efficiently processing the edits. First, we explain three important notations that we will use in our algorithm description. The cumulative edit list for version q+a

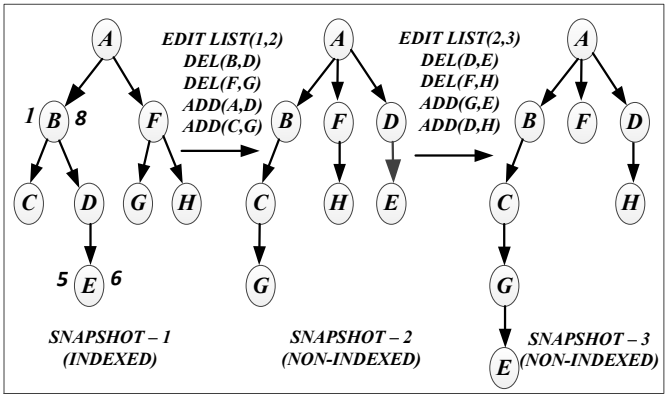

Figure 2.4: Illustration of SCISSOR framework

(denoted as `CEDList(q+a)`) contains all the edits occurring between the snapshot q (temporally closest indexed snapshot) and snapshot (q+a). The `Current_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 2 shows the pseudo-code of our algorithm. In the initialization phase of the algorithm (lines 1-19 of the algorithm), 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)`). In the former case (w was added between snapshots q and (q+a)), `Current_Reachability_Status` is set to FALSE, `ImList(v, w)` is initialized to include just w (please see lines 4-7) 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 `Current_Reachability_Status` set to FALSE (see 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 `Current_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 `Current_Reachability_Status` and the current composition of *ImList(v, w)*. Suppose `Current_Reachability_Status` is TRUE at the beginning of the i^{th} iteration the algorithm performs 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, `Current_Reachability_Status` is set to FALSE and the edit itself is removed from `CEDList(q+a)`.

If `Current_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 `Current_Reachability_Status` remains as FALSE. If, on the other hand, z is reachable from v, `Current_Reachability_Status` is set to TRUE. We also append z and all the parents of z until 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. `Current_Reachability_Status` is FALSE and there are no `Add` edits involving the vertices in `ImList`; or

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

Upon termination, the value of the `Current_Reachability_Status` holds the result of *SS-Reach(v, w, q+a)*. Note that the value of `Current_Reachability_Status` may flip multiple times during the algorithm.

We now illustrate our algorithm on the TEH shown in Figure 2.4 as it evolves through snapshots

1 through 3 (only snapshot-1 is indexed). Suppose we need to answer the query `SSReach(F,E,3)`. Snapshot-1 is the nearest indexed snapshot, and hence it is used as the initialization point. `Current_Reachability_Status` is set to FALSE as E_{pre} is not in between F_{pre} and F_{post} in the index corresponding to Snapshot-1 which means E is not reachable from F. 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,B,A`}. Since `Current_Reachability_Status` is FALSE 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 added by a ADD entry in CEDList(3). E is the leftmost vertex to have a ADD edit (`ADD(G,E)` is the corresponding edit). However since E already has a parent vertex, namely D, the entry `DEL(D,E)` should exist in CEDList(3) so as to maintain the "tree" property. This edit is found and processed along `ADD(G,E)`. As a result the ImList is modified to {`E, G, F, A`} and `Current_Reachability_Status` is set to TRUE (because E is reachable from F). The processed edits, `DEL(D,E)` and `ADD(G,E)` are removed from CEDList(3). In the second iteration where `Current_Reachability_Status` is TRUE, the algorithm scans the ImList from left to right looking for a vertex for which an edit in CEDList(3) deletes an inbound edge. The edit `DEL(F,G)` is found and processed. ImList is truncated to {`E, G`} and `Current_Reachability_Status` is set to FALSE. The entry `DEL(F,G)` is removed. In the third iteration as `Current_Reachability_Status` is FALSE (E is not reachable from F), the algorithm scans ImList for a vertex for which an edit in CEDList(3) adds an inbound edge. `ADD(C,G)` is found and processed. As a result ImList is modified to {`E, G, C, B, A`} and `ADD(C,G)` is removed from CEDList(3). Note however that `Current_Reachability_Status` still remains FALSE because E is not reachable from F. In this iteration, the algorithm stills scans the ImList from left to right looking for a vertex for which there is a corresponding ADD entry in CEDList(3) that adds an inbound edge. Since there are no such entries currently on CEDList(3), the algorithm terminates with the answer to the query `SSReach(F,E,3)` being FALSE (the last value carried by `Current_Reachability_Stat`

Size	Index No Snapshot (INS)		Index All SnapShots (IAS)		SCISSOR (1/100)	
	Indexing Time (sec)	Query Time (msec)	Indexing Time (sec)	Query Time (msec)	Indexing Time (sec)	Query 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 2.1: Indexing and Querying latencies of INS, IAS and SCISSOR

us). 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.

2.4.3 Correctness Proof

We have formally proved the correctness of our reachability algorithm. Our proof is composed of two main parts. First, we prove that we correctly processes each edit. In other words, does our algorithm update the state variables (Current_Reachability_Status, ImList, CANLists and CEDList) correctly at each iteration of the algorithm? Second, we prove that although our algorithm processes a subset of edits in the CEDList, it will not *miss* any edit in the CEDList that could have changed the final outcome of the reachability query. In other words, we will prove that any edits that are not processed by our algorithm would have had no effect on the final outcome.

Our proof utilizes two related concepts called superseding edit and dominant edit which are defined as follows. An edit E1 is said to supersede by another edit E2 in the i^{th} iteration of the algorithm, if E1 and E2 are both candidate edits in the this iteration (i.e., the destination vertices of both E1 and E2 are in the ImList) *and* the destination-vertex of E1 appears left to the target-vertex of E2 in ImList(see Section 4.1). An edit E1 is said to be dominant in the iteration i if its destination vertex appears in the ImList at the beginning of the iteration i and E1 is not superseded

by any other edit in the iteration.

The following three lemmas capture the most important aspects of our proof.

Lemma 1: If the invariants with respect to the ImList (see Section 2.4.1) holds true at the beginning of an arbitrary iteration i, these invariants will remain true after the completion of the iteration i irrespective of whether add edit or delete edit was processed in the iteration.

Lemma 2: An arbitrary add edit `ADD(x, y)` in the CEDList does not have any effect on the query SSReach(v, w, q+a) if *both* of the following conditions are not simultaneously satisfied in any single iteration of the algorithm – *(Condition-1)*: The edit is the dominant at the beginning of the iteration *(Condition-2)*: the `Current_Reachability_Status` at the beginning of the iteration is FALSE. Similarly, a delete edit `DEL(x, y)` in the CEDList does not have any effect on the query SSReach(v, w, q+a) if *both* of the following conditions are not simultaneously satisfied in any single iteration of the algorithm – *(Condition-3)*: The edit is the dominant at the beginning of the iteration; and *(Condition-4)*: the `Current_Reachability_Status` at the beginning of the iteration is TRUE.

Lemma 3: The effect of non liner processing of edits while answering the queries is same as linear processing of the edits when the following conditions are satisfied – *(Condition-1)*: All order dependent edits are processed in the same order as they appear in the edit list, where an arbitrary pair of edits (*DEL(v,w)* and *ADD(x, y)*) are order dependent if they operate on same edge. *(Condition-2)*: A nonexistent edit will never be processed while answering the queries.

Proof: Suppose the edit does not satisfy Condition-1 in any iteration when the `Current_Reachability_Status` is FALSE at the beginning of the iteration. This can happen because of two scenarios. First, the vertex y does not appear in ImList in any iteration where `Current_Reachability_Status` is FALSE. An edit that adds a new edge by its very nature creates new paths. However, since y is not on the ImList in any iteration when `Current_Reachability_Status` is FALSE, there is no existing path from y to w in any of these iterations. This implies that any new path that is created as a result of this edit does not pass through w. Thus we can safely

say in this scenario, the edit Add(x, y) has no effect on the final outcome. Now let us consider the second scenario – y appears in the ImList in one or more iterations when `Current_Reachability_Status` is FALSE, but in each of those iterations it is superseded by another edit. Suppose the ImList at the beginning of the i^{th} iteration is {w, u, r, . . . , y} and the `Current_Reachability_Status` is FALSE. Suppose there is another edit Add(q, u) in CEDList in addition to Add(x, y). Since y appears to the right of u, Add(q, u) supersedes Add(x, y) and our algorithm chooses to process to process Add(q, u). We will now show that even if he had processed Add(x, y) in this iteration, its effect on the final outcome is nullified. Notice that since each snapshot is guaranteed to be a tree, Add (q, u) is always accompanied by the edit Delete(u, r) (to ensure that node u has only one incoming edge. Suppose we process Add(x, y) in the i^{th} iteration. If the vertex y is reachable from v, the ImList will be updated to {w, u, r, . . . , y, x . . . , v} and the `Current_Reachability_Status` becomes TRUE at the end of this iteration. If, on the other hand, y is not reachable from v the ImList will be updated to {w, u, r, . . . , y, x, . . . , z} where z is the root of the y's tree and `Current_Reachability_Status` remains as FALSE. Let us consider the first scenario – `Current_Reachability_Status` is TRUE and ImList is {w, u, r, . . . , y, x . . . , v} for the $i+1^{th}$ iteration. Now in this iteration, we look for Delete edits involving any of the vertices in the ImList. Clearly, there is at least one such edit Delete (u, r). If no other Delete edits that supersedes this edit, we process this edit in the $i+1^{th}$ iteration which breaks the path from w to y. ImList is updated to {w, u } and `Current_Reachability_Status` is set to FALSE. Thus this Delete edit nullified the effects of Add(x, y). If, on the other hand, at the beginning of the $i+1^{th}$ iteration `Current_Reachability_Status` is FALSE and {w, u, r, . . . , y, x, . . . , z}. Now we look for Add edits involving the vertices currently in the ImList. Now, Add (u, r) is one such edit. Assuming that this edit is not superseded by other edits, it will be chosen for processing. However notice that the vertex u already has a parent. Thus, we also have to process the edit Delete (u, r). This edit breaks the path (created by the Add (x, y)) from y to v, thereby nullifying the effect of Add(x, y). Thus, in both scenarios, processing Add(x, y) has no effect on the final outcome when

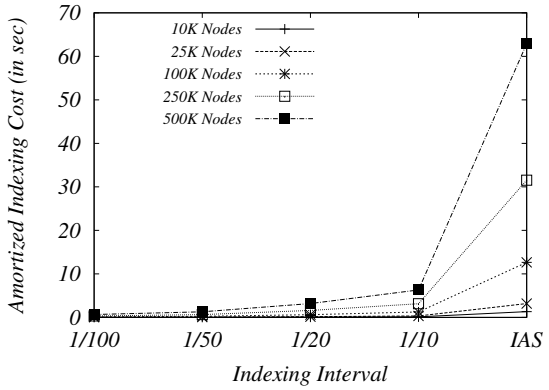

Figure 2.5: Amortized Indexing Cost (varying vertices)

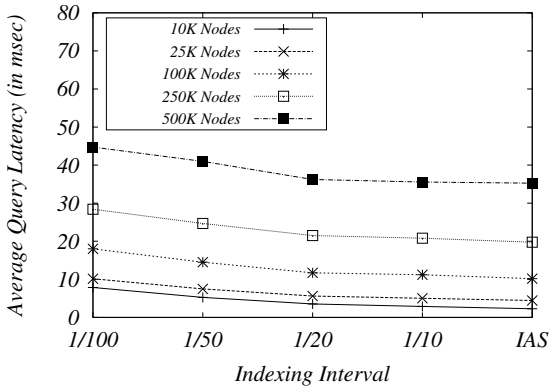

Figure 2.6: Avg Query Latency (varying vertices)

there is a superseding edit. Finally, let us consider the case when, Add(x, y) is on the ImList and is the dominant edit in one or more iterations, but in each of those iterations `Current_Reachability_Status` is TRUE. As we mentioned before, Add edits by their very nature, create new paths and increase connectivity. But they will not cause reachable vertex pairs to become unreachable. Since, in each iteration when Add(x, y) is dominant, w is already reachable from v, processing the edit Add(x, y) will not alter the final outcome of the algorithm.

By proving the correctness and completeness of edit processing, we have shown that our algorithm for snapshot-specific reachability testing is correct.

2.5 Indexing Decision Component

We now describe the second novel component of the SCISSOR framework namely a heuristic-based technique to decide which snapshots to index. Specifically, ours is an online scheme that decides whether to index an incoming snapshot. A good strategy in this regard should provide the following two features. First, it should manage and optimize both of the key performance factors,

namely, indexing costs and query performance. Second, it should provide flexibility to the TEH application to tradeoff one for the other (e.g., achieve better query performance at the cost of higher indexing overheads or vice versa). In other words, the TEH application will be able to *tune* the tradeoff between indexing overheads and query performance.

A straightforward approach for selecting snapshots for indexing is *periodic indexing* (e.g., selecting every k^{th} snapshot for indexing). Although this is a simple strategy, it may not be very effective managing indexing costs and query performance. This is because, hierarchies may change slowly during some snapshot sequences whereas they may experience rapid changes during other snapshot sequences. Periodic indexing incurs the same indexing cost irrespective of the rate and the extent of changes of the hierarchy.

A second option in this regard will be to use number of edits since last indexed snapshot as the criterion to decide whether to index an incoming version. If the cumulative number of edits since last indexed snapshot exceeds a certain threshold (which may be specified by the application) than the incoming snapshot is indexed. The problem with this approach however, is that it does not consider the effects of individual edits on the structure of the hierarchy. Some edits may have only minor effects on the structure of the hierarchy (and thus may affect only few queries) while other edits may drastically alter the hierarchy (thus affecting significant number of queries). Deletion of the edge (v, u) where u is a leaf in a shallow branch (i.e., the distance between u and the root is small) is an example of a minor edit whereas an edge deletion that results in disconnecting a large subtree from the original hierarchy will drastically alter the hierarchy.

Towards overcoming the limitations of the above approaches, our technique considers the cumulative effect of all the edits that have occurred since the last indexed snapshot on the reachability statuses of arbitrary pairs of vertices. Specifically, for each edit we quantify the fraction of all possible reachability queries that can be affected by the edit. Let us suppose the hierarchy has N vertices. The total number of unique reachability queries for this hierarchy is ${}^{N}\mathrm{P}_2 = N \times (N-1)$. Let us consider the edit `DEL(u, v)`. This edit will affect the reachability from any vertex that is

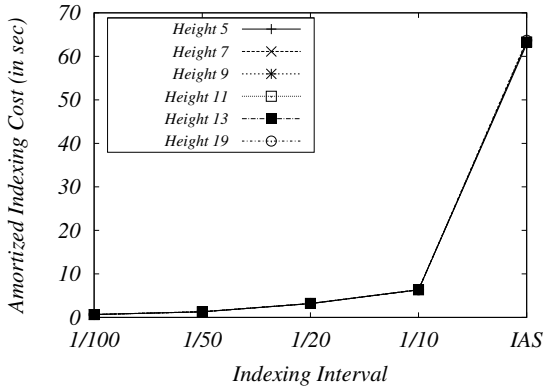

Figure 2.7: Amortized Indexing Cost (varying height)

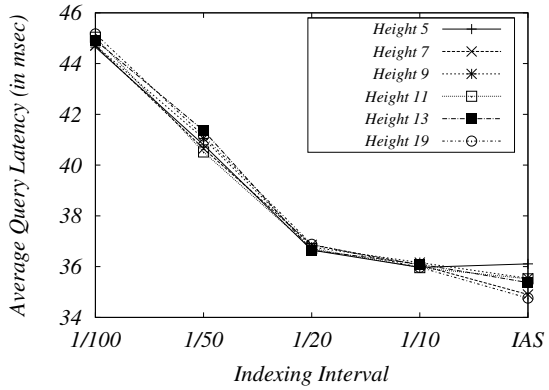

Figure 2.8: Avg Query Latency (varying height)

an ancestor of u (including u) to any vertex that is a descendant of v (including v). Therefore, the fraction of unique queries that will be impacted by this edit is represented as `Fq(DEL(u,v))` is $\frac{\mu(u)\times\lambda(v)}{N\times(N-1)}$, where $\mu(u)$ indicates the number of ancestors of u (including itself) and $\lambda(v)$ indicates the number of descendants of v (including v). The cumulative effect of all the edits that have occurred since the last indexed snapshot can now be approximated as the sum of the Fq values of each of those edits. If this sum of Fq values exceeds certain application specified threshold ρ, the incoming snapshot is selected for indexing. Note that TEH applications can tune the tradeoff between query efficiency and indexing costs by appropriately setting the threshold ρ. Higher values of ρ signifies lower indexing costs and vice-versa.

2.6 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 3.1, 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.

We have evaluated SCISSOR on a server with 3.10GHz quad-core 64-bit processor and 8 GB RAM that runs Ubuntu. MySQL server has been used for storing the pre and post-order index values as well as the edits.

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 the first 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 2.1 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

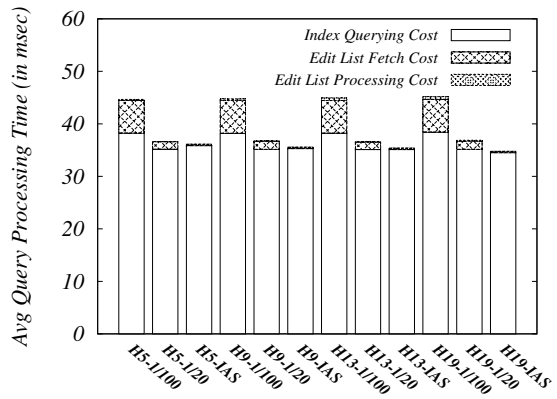

Figure 2.9: Split Query Processing Time (varying height)

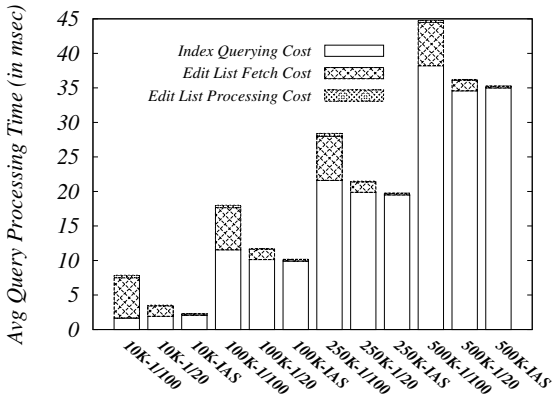

Figure 2.10: Split Query Processing Time (varying vertices)

a 500K node TEH) with marginal increase ($< 12\%$ for the same TEH) in query latencies when compared with the other approaches. In following experiments we show how SCISSOR balances the query and indexing latencies.

Figure 2.5 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 linearly 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 2.6 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 edits that needs

to be processed.

In order to further clarify the point regarding query latencies, stacked histogram in Figure 2.10 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 2.10 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.

In Figures 2.12 we show the cumulative distribution function (CDF) of the number of edits in the EDList that are actually processed by our algorithm when answering the reachability queries in our workload. The size of the TEH is 10K vertices. The figure shows the results when the fraction of indexed snapshots is set to $\frac{1}{100}$, $\frac{1}{50}$, $\frac{1}{20}$ and $\frac{1}{10}$. It can be seen from the figure that for all settings of indexed snapshot fractions, all the reachability queries were answered by processing only 5% of the edits in the corresponding EDList. We can also observe that, as the fraction of snapshots indexed increases more queries are answered without processing any edits. Figure 2.13 shows the CDF on a TEH of 250K vertices. These results demonstrate that our algorithm, answers the queries by processing only the edits that are likely to impact reachability status by avoiding unnecessary overheads.

In results shown in Figures 2.7, 2.8 and 2.9, we vary the height of the TEHs. All the TEHs considered for these experiments have 500K vertices but their heights range from 5 to 19. Figure 2.7 shows the amortized indexing costs as the indexed snapshot frequency varies from $\frac{1}{100}$ to 1 (which corresponds to IAS). The results show that height of the TEH has very little effect on the indexing costs. IAS again has highest indexing costs and INS has zero indexing costs. Figure 2.8 indicates the average query latency. Again, we see that height of the TEH has very little impact. The query latency increases as smaller fractions of snapshots are indexed because of edit

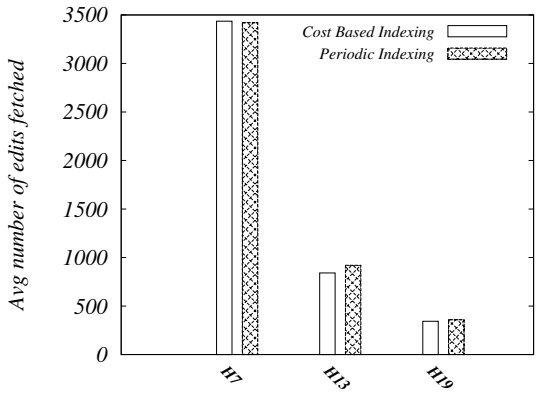

Figure 2.11: Cost Based Indexing vs Periodic Indexing

list processing overhead.

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.

Benefits of Cost Based Indexing: In this experiment, we study the benefits of our cost-based indexing strategy over periodic indexing. For this experiment, we consider three hierarchies of 500K nodes each but with varying heights (7, 13 and 19). We index each hierarchy using cost-based indexing as well as periodic indexing. For each experiment, we ensure that the same fraction of snapshots are indexed by both schemes (in order to ensure fairness in terms of indexing costs). Figure 2.11 shows the comparison of average number of edits fetched per query for answering the queries. Cost based indexing fetches requires lesser number of edits to be fetched compared to periodic indexing mechanism, indicating that cost-based indexing selects better snapshots to index.

To summarize, through our experimental study we have demonstrated that the SCISSOR framework balances indexing costs and query processing latencies efficiently.

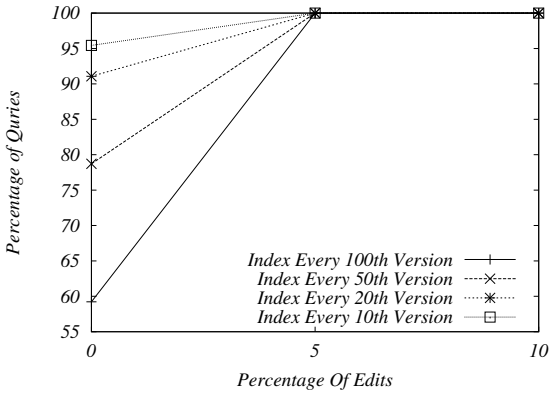

Figure 2.12: CDF for 10K Vertices

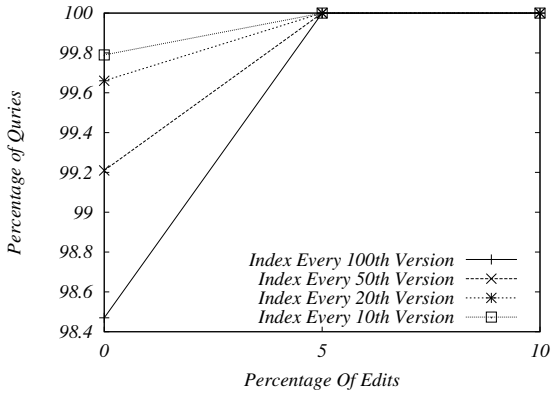

Figure 2.13: CDF for 250K Vertices

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

$CEDList \leftarrow Initialize\ with\ edits\ between\ q\ and\ q + a$
$CEDList \leftarrow Pre - process\ (CEDList)$
if w not in snapshot q **then**
  if $CEDList$ contains $Add(w)$ **then**
    $ImList \leftarrow w$
    $Current\_Reachability\_Status \leftarrow false$
  else
    $Current\_Reachability\_Status \leftarrow false$
    exit
  end if
else
  if $v^q_{pre} < w^q_{pre} < v^q_{post}$ **then**
    $Current\_Reachability\_Status \leftarrow true$
    $ImList \leftarrow fetchAncestors(v, w)$
  else
    $Current\_Reachability\_Status \leftarrow false$
    $ImList \leftarrow fetchAncestors(w)$
  end if
end if
while i ¡ Size of $ImList$ **do**
  $vertex \leftarrow ImList[i]$
  if $Current\_Reachability\_Status == TRUE$ **then**
    $delete\_edit \leftarrow SearchCEDList(vertex)$
    if $delete\_edit$ is not $null$ **then**
      $Imlist \leftarrow process(delete\_edit)$
      if $Imlist$ doesn't contain v **then**
        $Current\_Reachability\_Status \leftarrow FALSE$
      end if
      $CEDList \leftarrow remove(CEDList, delete\_edit)$
    end if
  else
    $add\_edit \leftarrow SearchCEDList(vertex)$
    if add_edit is not $null$ **then**
      $Imlist \leftarrow process(add\_edit)$
      if $Imlist$ contains v **then**
        $Current\_Reachability\_Status \leftarrow TRUE$
      end if
      $CEDList \leftarrow remove(CEDList, add\_edit)$
    end if
  end if
  $i++$
end while

Chapter 3

Application Of SCISSOR: Scalable XPath Evaluation On Continuously Evolving XML

3.1 Overview and Background

CEXML is important for representing information in domains that are temporally dynamic. A CEXML document series (CEXML document, for short) consists of multiple versions of an XML document as it evolves. Examples of CEXML documents include country profiles (such as those in CIA World Factbook), employee information systems, and reporting structures of public and private organizations. In these domains, updates to the underlying information gives rise to new versions of XML documents. The document repository stores each version of every CEXML document. In CEXML document repositories, it is often necessary to query some specific version (which may be a past version) of a given CEXML document. For instance, consider a CEXML document series representing the organization structure of a company. In such a CEXML document, one of the queries might be to find the reporting chain of a particular employee on some specific (possibly previous) date. For evaluating such queries, we need to evaluate XPath expressions on specific version of the CEXML document. We refer to such queries as *version-specific*

CEXML queries.

3.1.1 Modeling CEXML Document

Consider a document V. Let $\{V_1, V_2, \ldots, V_q, \ldots, V_r\}$ be the different versions of the XML document. Without loss of generality a query related to a time period can be mapped to a specific version of the document. Hence, here after we specify queries related to a particular version of the document instead of time period. Let Diff(V_q, V_{q+1}) represent the changes occurring between versions V_q and V_{q+1}. A new document V_{q+1} can be obtained by applying Diff(V_q, V_{q+1}) on V_q. Diff(V_q, V_{q+1}) includes the edits performed on different types of nodes (e.g., element, attribute, content/text, comment etc.) present in the XML document. An edit involves creating, deleting or updating these nodes. We assume that Diff(V_q, V_{q+1}) between any two versions does not remove the attribute id which is the unique identifier of the record in the XML document. Let V_1 be the first version of the document and V_r be the most recent version. Our framework discussed in this chapter can answer XPath queries on V_q where $1 \leq q \leq r$.

3.1.2 Version-Specific XPath Expressions (VS-XPath)

An XML document consists of different types of nodes (e.g. elements, attribute, text/content, comment, processing instruction). XPath expressions select nodes or node-sets in an XML document. These expressions specify document traversal via two parameters: a set of context nodes and a list of steps, where a step consists of an axis, node and predicate. Following is the syntax of step in a static XML document. *axis::node-test[predicate]*. An axis defines the relationship between the selected nodes and context node. Node-test identifies a node from the set of selected nodes and predicate helps in further refining the node set. XPath axes that are of particular interest to us are: descendant, ancestor, following, and preceding (henceforth referred to as primary axes). All the remaining axes are a subset of these axes.

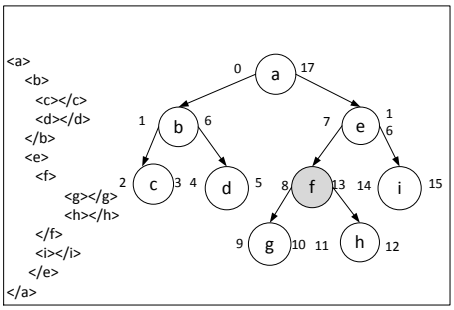

Figure 3.1: Interval Based Indexing Scheme

We define version-specific XPath expressions on a CEXML document in this section. An example of such an expression is provided below.

$$//employee[@id =' 101']/descendant :: salary : 10 \tag{3.1}$$

Version-specific XPath expression consists of two parts of which first part ($//employee[@id =' 101']/descendant :: salary$) represents a basic expression. Second part provides the document version number that needs to be queried. In this case document version number is 10. The above mentioned query determines the salary of an employee with id 101 in 10th version of the document.

3.1.3 Indexing XML Documents

Several techniques have been proposed to index XML documents of which interval based indexing is the most widely adopted technique. Figure 3.1 consists of XML markups along with its corresponding hierarchy. The hierarchical representation of XML illustrates the interval-based indexing scheme [75, **?**]. The number to the left of each vertex is its pre-order value whereas the number to its right is its post-order value. We leverage these indices to evaluate XPath Axes as explained below.

In Figure 3.1, let f be a context node (node against which an XPath expression is evaluated),

Axis Name	Mechanism	Result
descendant(f)	$pre(f_d) > pre(f)$ and $post(f_d) < pre(f)$	$\{g, h\}$
ancestor(f)	$pre(f_a) < pre(f)$ and $post(f_a) > pre(f)$	$\{a, e\}$
preceding(f)	$\{pre(f_p) < pre(f)\}$ - $\{post(f) < post(f_p)\}$	$\{b, c, d\}$
following(f)	$\{pre(f_f) > pre(f)\}$ - $\{post(f_f) < post(f)\}$	$\{i\}$

Table 3.1: Answering XPath Axes Using Interval Based Indexing

pre(f)= 8 and *post(f)=13* represent the pre index and post index values of context node f. Let us assume that f_d, f_a, f_p and f_f be a descendant, ancestor, preceding and following nodes respectively of a given context node(f). Second column in the Table 3.1 demonstrates the mechanism necessary for a node in the XML document to be an *ancestor*, *descendant*, *following*, *preceding* node of given context node(f). Third column in the Table 3.1 provides the set of nodes representing the corresponding axis.

3.1.4 Naive Approaches and their Drawbacks

A straightforward approach for answering VS-XPath queries on a CEXML document is to index every version of the document by using interval-based indexing and to use the conditions described in Table 3.1 to answer the queries. However, there are many drawbacks to this simple approach. First, the computational overheads of indexing every version is exorbitant. Second, the storage overheads are going to be high as well because of the need to save the index values of every version. Third, with this approach, the applications will have no control over the indexing costs vs. query efficiency trade-off. In other words, applications cannot *tune* the system to incur less indexing overheads even when they can tolerate small increase in query latencies.

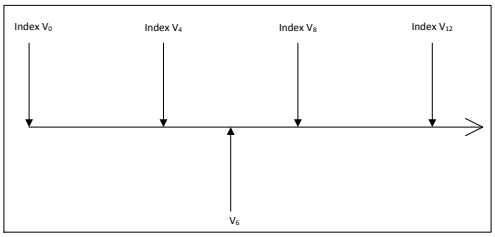

Figure 3.2: Framework Idea

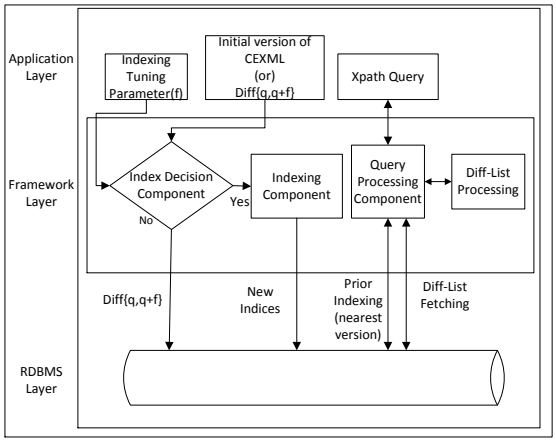

Figure 3.3: Architecture of CEXML Query Framework

3.2 Architectural Overview

The main idea is to selectively index interspersed versions of the XML document as shown in Figure 3.2. In Figure 3.2, the XML document has 12 versions including the initial version of which every 4th version is indexed. These subsets of versions are henceforth referred to as indexed-versions. These indices together with the list of changes occurring between versions will be used for answering XPath queries on any version of the XML document. An important question to address when designing the framework – *How to answer XPath queries on a particular version (for example, on version 6), especially if the version is not indexed?*. Before we address this question, we give a high-level overview of the framework.

Figure 3.3 shows the high-level architecture of our framework. The applications using our

framework give us the complete structure of the initial version of the Continuously Evolving XML document. For each subsequent new version, the application just provides the changes between the previous version and the current version in the form of an edits list. The edit list between version q and version $(q+1)$ is represented as Diff$(q, q+1)$. Any edit on a XML document can be converted to a set of add and $delete$ edits where `Add(e(t),rid,vid)` replaces the contents of an element e with text t of a record with id rid in the version vid of the document or creates the element in that record if it does not exist and `Delete(e,rid,vid)` indicates the removal of an element e from a record with id rid in version vid of the document.

The application layer provides a *indexing tuning parameter(f)* to index decision component which is part of our framework layer to decide whether the next version of the document needs to be indexed or not. If the decision is to index the next incoming version then the edit list of incoming version along with the previous version are sent to *indexing component*, else the edit list between the versions is inserted into the database. XPath queries issued by applications are handled by *query processing component* in the framework layer. The detailed methodology of the evaluation is discussed in the later section.

3.2.1 Indexing Component

The first step in evaluating XPath queries using our framework is to periodically index the versions. Suppose we have n versions of the document and we decide that we are going to have a frequency of f then we are indexing $(*\frac{n}{f})$ versions where $f < n$. In order to periodically index a CEXML document we use a combination of SAX and DOM parsers. It is done in two phases as mentioned below.

In the first phase, we parse the document (version *q*) using the SAX parser to assign pre and post order indices to the nodes (element, text, etc.) in the document as shown in Figure 3.1 and save the indices in to database. In the second phase, we process all the edits happening between the indexed-version(q) and the new version that needs to be indexed(q+f, where f being the frequency

of indexing) using the DOM parser to get an updated version $q + f$. Later, Update q ($q \leftarrow q + f$) and repeat the above steps until $q + f \leq n$

3.3 XPath Query Evaluation

Without loss of generality the query evaluation is formalized as follows.

Suppose versions `v` and `(v+b)` are indexed (i.e., pre-order and post-order indices are available for version `v` and version `(v+b)`). Also the edit list for all intermediate versions between `v` and `(v+b)` are available. The problem now is to evaluate the primary axes namely ancestor, descendant, preceding or following on CEXML document. For example, if the query asks us to find all the ancestors of element e in record with id rid in version v of the XML document then the syntax of the query is as follows,

```
//e//ancestor::nodetest[@id='rid']:v
```
(3.2)

In the above mentioned query as the version v is indexed, we find all the ancestors of the element e using interval based indexing technique provided in Table 3.1. Let us assume the query is on version $(v + a)$ where $0 < a < b$, which is a non-indexed version. In order to answer the query on a non indexed version$(v + a)$ we have to consider the edits happening between the previous indexed-version(v) and queried version$(v + a)$. However, the question is *how to come up with a method for efficiently filtering the edits occurring between versions v and (v+a)?* A trivial way is to process *all* edits occurring between versions v and $(v + a)$ in the order they appear in the edit list, and analyze the cumulative effect of these edits on that version to answer the query. 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 XML document), and processing them adds unnecessary overheads. Second, processing an edit requires loading the corresponding part of the XML tree and updating it as per

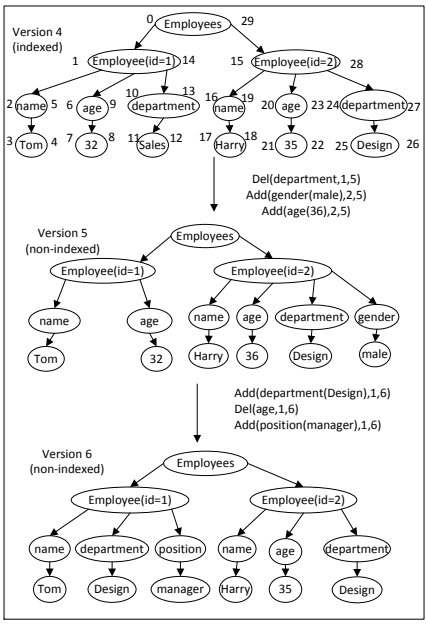

Figure 3.4: Illustration of the Framework

the edit. Thus, processing every edit imposes high processing overheads.

3.3.1 Observations

In order to avoid these limitations we need to come up with a effective method to *analyze only those edits which effect the part of the XML document as the document evolves from version v to version v+a*. In order to achieve this first we need to know about the behavior of an XML document to filter out all the unnecessary edits that do not effect the element e in the above mentioned query. We make the following observations on the behavior of an XML document that help us to filter out the edits. In order to understand the nature of the XPath axes Figure 3.4 is taken as a running example for the rest of the chapter.

Observation 1: If we consider an ancestor query, any edits happening on the context node or its parent nodes are considered because each record is independent and any edit happening on other records besides the record which contains the context node do not effect the result of the query.

Hence, all the edits happening below the context node in an XML tree are not processed. For example, in Figure 3.4, if an ancestor query is asked on $department$ in $Employee(id = 1)$ record in version 4. Then we do not consider the edits happening on $Employee(id = 2)$ because they do not effect the query, but we consider edits happening on $Employee(id = 1)$ since the query is on this record. We can further filter out the edits on record $Employee(id = 1)$ because edits on the other elements do not effect outcome of the query but edits on the element $department$ effect the result, hence we consider only those edits.

Observation 2: Similarly, if we consider a descendant query, any edit happening on the nodes above the context node in an XML tree need not be processed since these edits do not effect the query, but we consider edits happening only on the context node or its child nodes. For example, in Figure 3.4, if a descendant query is asked on $department$ in $Employee(id = 1)$ record in version 4. Then we do not consider the edits happening on $Employee(id = 2)$ because they do not effect the query but we consider edits happening on $Employee(id = 1)$ since the query is on this record. We can further filter out the edits on record $Employee(id = 1)$ because edits on the other elements do not effect the query but edits on the element $department$ or on $Sales$ effects the result, hence we consider only those edits. The worst case scenario for this type of query would be when the descendants of the root element are asked, which results in all the edits being processed.

Observation 3: If a query is related to a preceding axis, any edit happening on the nodes after the context node in an XML tree need not be processed, since according to definition edits that happen the nodes below the context node do not affect the query. For example, in Figure 3.4, if a preceding query is asked on $department$ in $Employee(id = 1)$ record in version 4. Then we consider all the edits happening on the elements in $Employee(id = 1)$ except $department$ since it is he context node but do not consider the edits happening on $Employee(id = 2)$ because they do not effect the result of the query.

Observation 4: If a query is related to a following axis, any edit happening on the nodes before the context node in an XML tree need not be processed, since according to definition the nodes

above the context node are not part of the result. For example, in Figure 3.4, if a following query is asked on $department$ in $Employee(id = 1)$ record in version 4. Then we consider all the edits happening on the elements in $Employee(id = 2)$ but do not consider the edits happening on $Employee(id = 1)$ or on $department$ because according to definition the result of the query should only contain all the nodes that are after the context node $department$ and its descendants.

Hence, if the query mentioned earlier was on a non indexed-version, in order to get the final result first we need to get the ancestors of the element from the previous indexed-version v. Using the observations mentioned we filter out the edits, thereby we get the desired edits that effect the element that has been queried. Using these edits we process them on the ancestor list we initially had to obtain the final ancestor list.

3.3.2 VS-XPath Algorithm

We now outline the algorithm for evaluating XPath axes. The notations used in our algorithm are as follows: $DList$ in algorithm 2 contains the descendants list of the query, $PList$ in algorithm 3 contains the preceding list and $Diff$ contains all the edits ed_1, $ed_2 \ldots ed_n$ between the versions v and $v + a$. Algorithm 2 and 3 show the pseudo-codes of descendants and preceding algorithms, respectively.

In our algorithm for $descendants(e, id, v + a)$, lines 2-7 describe the process of how the query is evaluated when the queried version ($v + a$) is indexed. We search through all the nodes n in the document to see if they fall between the mentioned range and add them to the $DList$. Then in line 7 we return the final list of descendants of the query. In lines 8-23 we describe the process of obtaining the descendants if the query is on a non-indexed version. Lines 9-12 give us the initial list of descendants from the nearest indexed version, in our case we consider the previous version(v), and line 13 initializes the $Diff$ list for processing the edits on the initial list of descendants. Now lines 14-20 in the algorithm gives us the edits processing methodology on the $DList$. The edits are filtered based on the observations mentioned in earlier section, where if the edits are happening on

Algorithm 2 Descendants($e, id, v + a$)

1: $DList \leftarrow Initialize\ empty\ list$
2: **if** $v + a$ is indexed **then**
3: **for all** n such that **do**
4: $e_{pre}^{v+a} < n_{pre}^{v+a}$ and $e_{post}^{v+a} > n_{post}^{v+a}$
5: $DList \leftarrow Add\ n$
6: **end for**
7: return $DList$
8: **else**
9: **for all** n such that **do**
10: $e_{pre}^{v} < n_{pre}^{v}$ and $e_{post}^{v} > n_{post}^{v}$
11: $DList \leftarrow Add\ n$
12: **end for**
13: $Diff \leftarrow Initialize\ list\ with\ edits\ between\ v\ and\ v + a$
14: **for all** ed in $Diff$ **do**
15: **if** ed is in record with id and contains element e **then**
16: **if** ed is $delete\_edit$ **then**
17: $DList \leftarrow process(delete\_edit)$
18: **else**
19: $DList \leftarrow process(add\_edit)$
20: **end if**
21: **end if**
22: **end for**
23: return $DList$
24: **end if**

the same record id as in the query and if they contain the same element name e then we consider those edits, and if the edit is a $delete - edit$ then we process the edit by taking the element name in the edit and removing from the $DList$ and continue on to the next edit. Suppose the edit is an $add - edit$ then we add the element name present in the edit to the $DList$. This process goes on until all the edits have been processed to return the final list of descendants in line 23. Queries on ancestors follow a similar methodology but the way of obtaining the initial $DList$ changes where the range is replaced by the range mentioned in Table 3.1 to place the nodes in the list.

In our algorithm for $preceding(e, id, v + a)$, lines 2-11 describe the process of how the query

is evaluated when the version $v + a$ is indexed. We search through all the element nodes n in the document to see if they fall between the mentioned range and add them to the $PList$. In lines 7-10 we search through all the nodes that fall between the mentioned range and remove them from the $PList$. Finally, in line 11 we return the final list of preceding nodes of the query. In lines 12-31, we describe the process of obtaining the preceding nodes if the query is on a non-indexed version. Lines 12-20 give us the initial list of preceding nodes from the nearest indexed version, in our case we consider the previous version, and line 21 initializes the $Diff$ list for processing the edits on the initial list of ancestors. Now, lines 21-30 in the algorithm give us the edits processing methodology on the $PList$. The edits are filtered based on the technique mentioned before for preceding where if the edits are happening on the same record id as in the query and if they contain the same element name e or on any other record whose id is less than the id in the query, then we consider those edits. If the edit is a $delete - edit$ then we process the edit by taking the element name in the edit and removing it from the $PList$ and continue on to the next edit. Suppose the edit is an $add-edit$ then we add the element name present in the edit to the $PList$. This process goes on until all the edits have been processed to return the final list of preceding nodes in line 31. Queries on $following$ follow a similar methodology but the way of obtaining the initial $PList$ changes where the range is replaced by the range mentioned in Table 3.1 to place the nodes in the list.

To show how the framework works we consider an example query on well known employee schema. Let us consider that there are 8 versions of the document including the initial version, where every 4th version is indexed. The illustration of the framework for this example is given in Figure 3.4 and the query is as follows

$$//Employee[@id =' 1']/descendant :: department : 6 \tag{3.3}$$

Hence, the query can be comprehended as, we want the descendants of department of an employee with id 1 in version 6. Therefore, in order to answer this query first we need to find out

if the query was on indexed-version, if so it could be answered by using available indices. If the query is on non indexed version then we need to follow a different approach. First we need to get the descendant list of department from the previous indexed version, so the list is `[Sales]`. After we get the descendants list from the previous indexed-version we need to gather all the edits happening between version 4 and 6. The earlier mentioned filtering technique in section 3.3.1 for the descendants is used to filter the edits. For this example we consider that there were four edits on the employee record with id '1' that happened in between version 4 and 6. The edits are as follows

```
Del(department,1,5)
Add(department(Design),1,6)
Del(age,1,6)
Add(position(manager),1,6)
```

We filter the above mentioned edits to two since only the first two edits effect the element (department) that has been queried. The next step in the evaluation is to process the edits in chronological order on the descendant list obtained from previous indexed-version. Of the two edits, Del edit is processed first, which results in an empty list since the element has been deleted from the record, then the Add edit is processed which provides a list containing the following descendants `[Design]`. All the necessary edits happening between versions 4 and 6 are processed, hence the final list of descendants for the query is `[Design]`.

3.4 Experiments and Results

In this section, we discuss the experiments performed to evaluate the framework. We have implemented the framework in Java. The implementation is done in a modular fashion so that specific components can be enabled or disabled for evaluation. We have evaluated our framework on two setups to study its performance. First is on a system with 2.4GHz dual-core 64-bit i5 processor and

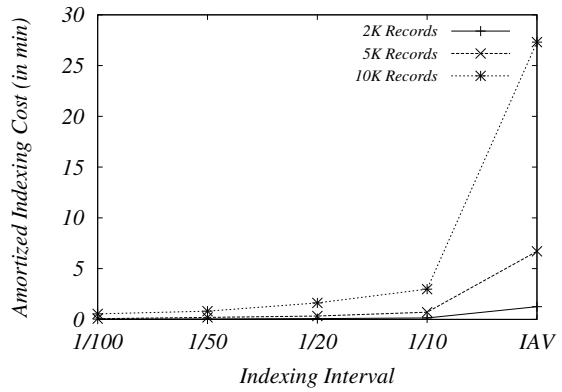

Figure 3.5: Amortized Indexing Cost on AWS-server

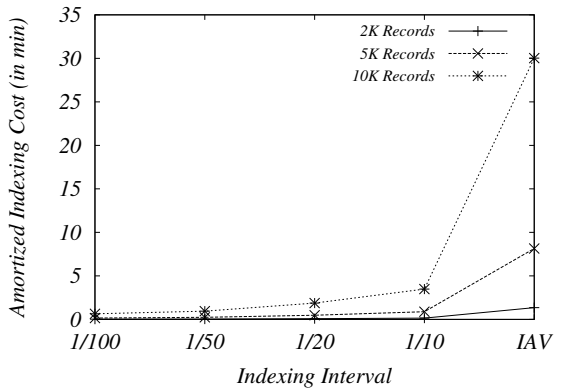

Figure 3.6: Amortized Indexing Cost on Windows-LT

8 GB RAM that runs Windows (henceforth referred to as Windows-LT) while another setup is on a Amazon EC2 instance which runs Windows server with eight-core 64-bit Xeon processor with 2GHz processing power per core and 8 GB RAM (henceforth referred to as AWS-server). MySQL server has been used on both setups for storing the pre and post-order indices as well as the edits.

In order to comprehensively study the behavior of the proposed framework, we use an XML schema related to employee information in our experimental evaluation. We have varied the number of records in the document from 2K to 10K to test its scalability. The total number of versions for the XML document in this experiment are 100. In order to create the versions, edits were randomly generated while each edit in an XML document can happen on either its content or markups. There are a total of 5000 edits, with 50 edits happening between each version of which half of them are add edits and the rest are delete edits. The query workload consists of 500 ancestor and descendant queries as well as 500 preceding and following queries which are randomly distributed across all the versions.

We evaluate the framework on three main performance aspects, namely indexing overhead, query latency and storage cost. We use *amortized indexing latency* as the metric for quantifying

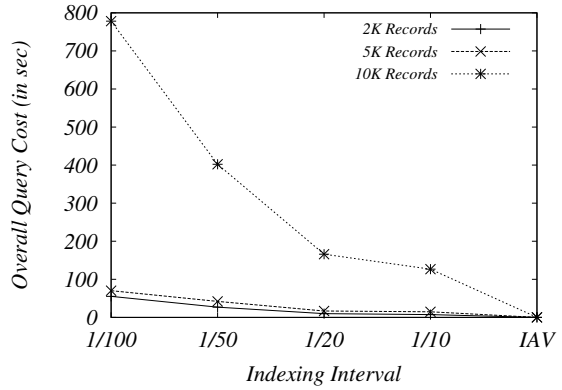

Figure 3.7: Avg Query Latency For Preceding/Following Queries on AWS-server

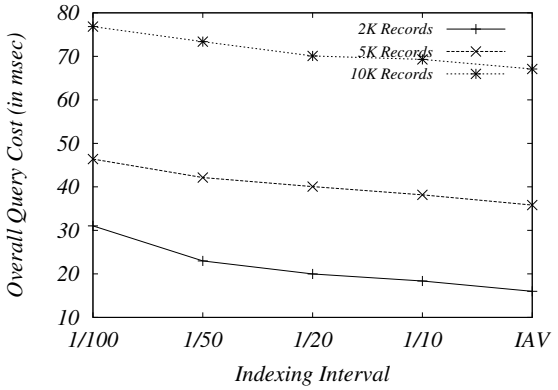

Figure 3.8: Avg Query Latency For Ancestor/Descendants Queries on AWS-server

indexing overhead. Suppose an XML document has n versions 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 the framework, 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. *Amortized storage cost* is the ratio of total disk-space needed to store indices and edits of all versions to number of versions of the XML document (n). We compare our framework with naive approach of indexing every version (henceforth referred to as Index All Versions (IAV)).

Figure 3.5 and Figure 3.6 indicate the amortized indexing costs by varying records from 2K to 10K in the document. The X-axis indicates, log-scale of the percentage of versions indexed in the framework where 1/100 means indexing every 100th version. IAV has highest amortized indexing cost compared to other framework schemes. As we have mentioned before, amortized indexing cost decreases almost linearly as we reduce the indexing frequency. This behavior is entirely due to the number of versions that needs to be indexed.

Figure 3.8 and Figure 3.9 show the average query latencies for the ancestor and descendant

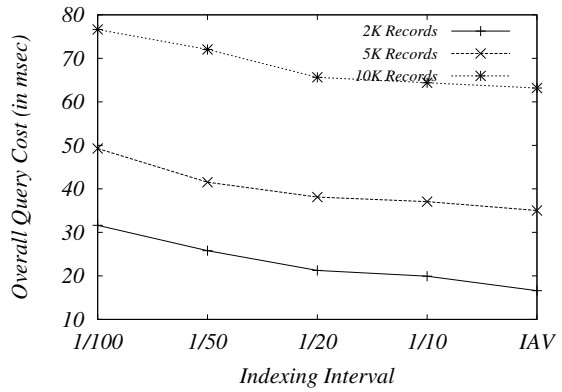

Figure 3.9: Avg Query Latency For Ancestor/Descendants Queries on Windows-LT

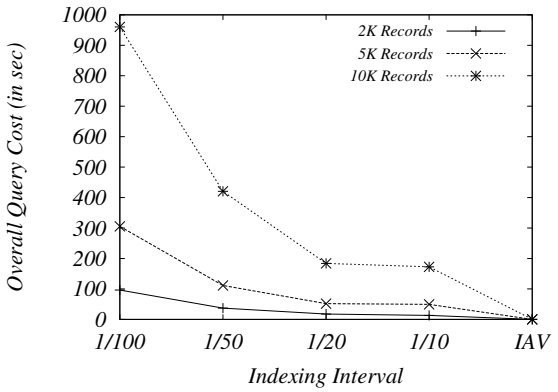

Figure 3.10: Avg Query Latency For Preceding/Following Queries on Windows-LT

XPath axes. The results show that when the fraction of indexed versions is set to 10 we have the least query latency among the framework schemes as the number of edits that need to be processed is less, whereas it has the highest query latency when the fraction is set to 100 because of the increase in the number of edits that needs to be processed. IAV has least query latency as no edits needs to be processed where the query can be directly answered using the available indices.

Figure 3.7 and Figure 3.10 show the average query latencies for the preceding and following XPath axes. The results show a similar trend as Figure 3.8 and Figure 3.9 but with an increase in the latency, this is because of the number of edits that are being processed increased significantly. The increase in edits is due to the nature of the queries as they cover larger sub set of nodes in the document in general.

In order to further clarify the point regarding query latencies, stacked histogram in Figure 3.11 and Figure 3.12 represents three types of query latencies, first is the latency involved in querying the nearest indexed version followed by the latency involved in fetching the edits between the nearest indexed version and the queried version 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

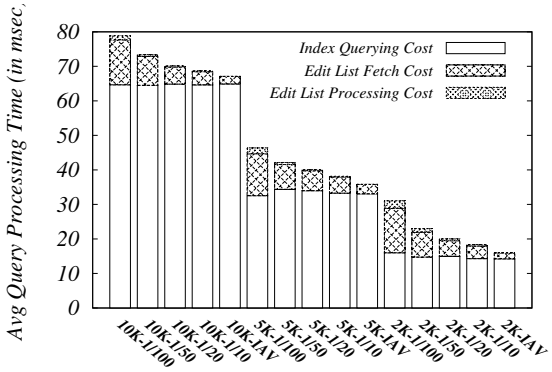

Figure 3.11: Split Query Processing Time For Ancestor/Descendant Queries on AWS-server

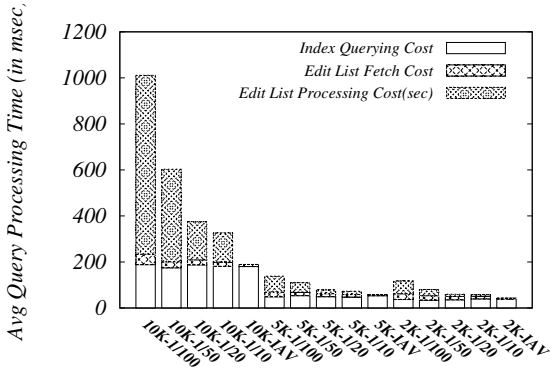

Figure 3.12: Split Query Processing Time For Preceding/Following Queries on AWS-server

of versions indexed (every 100^{th} version). The noteworthy point in Figure 3.11 is that edit list processing latency is a very minor contributor to the average query latency and in Figure 3.12, edit list processing latency is the major contributor to the average query latency because of the number of edits being processed. In general edit list processing latency increases as the fraction of indexed versions is reduced.

From the above experimental results we can observe that both the setups show a similar trend in the results. However, the main point to be noted is the computational costs between the two setups. We can see that the AWS-server has 15 percent less indexing costs compared to the Windows-LT setup.

Figure 3.13 reperesents the overall storage cost for storing the preorder and post order indices and the edits over all the versions of the XML document by varying the fraction of the versions indexed. As expected, overall storage cost increases almost linearly with the fraction of the versions indexed.

In summary, these experiments show that our framework yields significant reduction in indexing costs with only a marginal increase in query latencies.

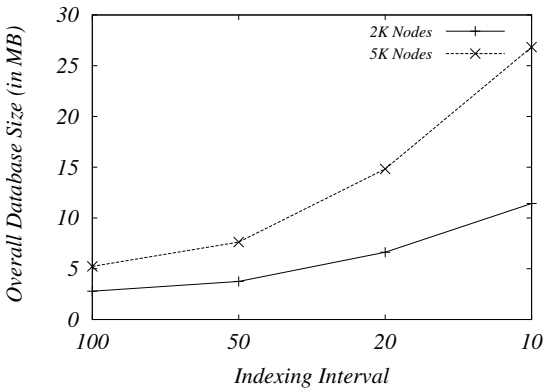

Figure 3.13: Database Storage Cost

Algorithm 3 Preceding($e, id, v + a$)

$PList \leftarrow Initialize\ empty\ list$
if $v + a$ is indexed **then**
  for all n such that **do**
    $e^{v+a}_{pre} > n^{v+a}_{pre}$
    $PList \leftarrow Add\ n$
  end for
  for all n such that **do**
    $e^{v+a}_{pre} > n^{v+a}_{pre}$ and $e^{v+a}_{post} < n^{v+a}_{post}$
    $PList \leftarrow Remove\ n$
  end for
  return $PList$
else
  for all n such that **do**
    $e^{v}_{pre} > n^{v}_{pre}$
    $PList \leftarrow Add\ n$
  end for
  for all n such that **do**
    $e^{v}_{pre} > n^{v}_{pre}$ and $e^{v}_{post} < n^{v}_{post}$
    $PList \leftarrow Remove\ n$
  end for
  $Diff \leftarrow Initialize\ list\ with\ edits\ between\ v\ and\ v + a$
  for all ed in $Diff$ **do**
    if ed is in record with id and contains element e or ed is in record with id $< id$ **then**
      if ed is $delete\_edit$ **then**
        $PList \leftarrow process(delete\_edit)$
      else
        $PList \leftarrow process(add\_edit)$
      end if
    end if
  end for
  return $PList$
end if

Chapter 4

CoUPE: Continuous Query Processing Engine for Evolving Graphs

4.1 Introduction

Graph is a fundamental data structure used for data modeling in many domains such as social networks, evolutionary genomics, communication networks etc. The graphs in many of these emerging domains are characterized by two properties – (1) they are massive (graphs with millions of vertices and edges are very common); and (2) they are dynamic (i.e., they change over time). For example, consider the relationship graph of a typical social network. Such a graph undergoes constant changes. For instance, a new vertex is added every time a new user joins the social network and the vertex corresponding to a user is deleted if the user exits the network. Similarly, edges are added and deleted when users friend or unfriend other users. Such graphs that change over time are also referred to as *Continuously Evolving Graphs* (CEGs, for short).

Due to the prevalence of CEGs in modern applications, a number of different types of queries have recently gained importance. These include reachability queries (which test whether a path exists between a given pair of vertices), patten matching queries (finding subgraphs that are similar

to a given query graph) and keyword queries (extracting vertices and edges related to a given set of keywords). Furthermore, since the graphs change over time, each of these queries can be further classified into several temporal categories such as snapshot-specific queries (queries that pertain to a particular version/time of the data graph), inverse snapshot queries (finding the first version/time when a particular condition became true) and continuous queries (trigger queries that need to be continuously executed as the graph undergoes changes). While efficient query execution in static graphs has been extensively studied [35, 80, 21, 22], queries on CEGs have received considerably less research attention [62, 61]. Techniques that have been designed for static graphs often do not work well for CEGs because many of these techniques execute queries from scratch and they are not designed to incrementally update previous results.

In this chapter, we focus on an important query called the *continuous reachability query* in CEGs. Consider a dynamic graph G. Let u and v be two arbitrary vertices in G. The reachability status from u to v at a given point in time is whether v is reachable from v at that time. A continuous reachability query seeks to continuously monitor the reachability status from u to v as G evolves and raises a trigger as soon as the reachability status undergoes a change. Continuous reachability queries are uniquely important in many applications. One of the application domains is social networking. Consider a facebook social graph that evolves over time. Let's assume that we are interested in finding if person B is reachable from person A through his friends via friend of a friend relationship and also let's assume that they are currently unreachable in the network. Over period of time person A adds a friend C, who also happen to have person$B$ as one of his/her friends. Our framework detects that person B is reachable from person A when person A adds C to his network.

A naive approach for answering reachability queries is to perform traversals (either in breadth or depth first orders) each time the graph undergoes a change. But this naive approach is prohibitively expensive. Researchers have shown that indexing the graphs on a relational database can help alleviate the performance problems for static graphs. Of the several indexing techniques

that have been proposed for this purpose [64, 80], interval based indexing [71] is among the most popular ones. In interval-based indexing each vertex is assigned a pre-order and a post-order index value, and these values are stored in a relational database. Reachability queries can be answered by testing whether the pre-order index value of the destination lies in-between the pre- and post-order index values of the source vertex.

While the interval-based indexing is very efficient for static graphs, it is not directly useful for answering continuous reachability queries as it does not capture the evolving nature of the graph. In this chapter, we propose a novel framework called **Co**ntinuous q**U**ery **P**rocessing **E**ngine (CoUPE) to answer continuous reachability queries in evolving graphs. In designing the CoUPE framework, we make three major research contributions:

- First, we introduce a generic indexing and querying framework for answering continuous queries in large time-evolving graphs. The central idea is to maintain indices for the evolving graph using a dynamic interval based indexing technique and reevaluate only subset of given queries that might get affected because of the structural change in the graph.

- Second, we design a highly efficient and scalable algorithm for updating the indices of graph by analyzing the changes happening on the graph.

- Third, we present a novel heuristic-based technique for deciding which subset of given queries are likely to get affected because of the most recent edit in the graph. We present this scheme as Selective Query Processing Scheme (SQPS).

We have also performed a number of experiments on large graphs to study the performance of the proposed techniques. Our experiments demonstrate that the techniques are highly efficient and they scale well.

4.2 Overview and Background

Continuous query evaluation is a different paradigm when compared to adhoc query evaluation (Eg: SQL queries, executed once to completion over the existing data set). In continuous query evaluation paradigm, queries are issued only once and then logically run continuously over the evolving data set. The initial state of the issued queries is evaluated based on the status of the current data set. Status of these queries changes as the underlying data set changes.

In the context of this work, we are interested in particular class of queries called reachability queries that checks if there exists a path between given pair of vertices in the graph. As the graphs evolves, a set of reachable queries might become unreachable because of edge deletions and a set of unreachable queries might become reachable because of edge additions. Our framework evaluates if any queries of interest are effected because of an edge addition/removal in the underlying graph.

Consider a graph G(V,E) that consists of set of vertices V and edges E at time instance T_1. Let Q represent a set of given queries of which Q_1 ($\subseteq$ of Q) are reachable. Let S be a stream of edits that modify the graph G(V,E). The processing engine can identify a set of queries Q_2 ($\subseteq$ of Q_1) that becomes unreachable because of an edit from edit stream S.

Figure 4.2 represents a directed acyclic graph. Let Q=$\{(A,B),(A,C),(D,G)(G,H)\}$ represent given query set of which (A,B) and (A,C) are reachable (Initial state of queries). Let there be an edit DELETE (A,C) in edit stream S which removes an edge from A to C. Once the edit is processed our framework identifies that (A,C) has become unreachable because of this edit.

4.3 Interval-based Indexing for Graphs

Breadth-first traversal, Depth-first traversal and Transitive closure are the earliest approaches for answering reachability queries in static structures. However, they do not scale well as we increase the size of the structure. An alternate approach that has been pursued in recent years is to maintain certain indexing information for the graph [71, 75, 80, 35, 64]. Interval-based indexing is one

such scheme that is very popular because of its simplicity, efficiency and its ability to provide good balance in the trade-off between indexing costs and query-time. It was originally proposed for tree structured data. Several researchers have proposed schemes that extend interval-based indexing for reachability testing in general directed graphs.

Figure 4.1 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. Every node in the graph is annotated with these values. As nodes in graph can have more than one incoming edge, single pair of values is not sufficient to encode the connectivity of the graph. Hence few nodes will get more than a pair of values as illustrated in Figure 4.2. A node `w` with $\mathtt{n}(> 1)$ incoming edges will have `n` pairs of values. An edge on which we reach `w` for the first time is referred to as a `tree edge`. We assign a pre-order value to node `w` and proceed to process its children. Once we process node `w`'s children, it receives a post-order value. Assume that we reach node `w` for the first time on edge $e_{\mathtt{i}}$ and later on edge $e_{\mathtt{j}}$, $(e_{\mathtt{j}} \neq e_{\mathtt{i}})$. We refer to $e_{\mathtt{j}}$ as a `non-tree edge`.

With this index in place, the reachability query that checks the reachability from `v` to `w` is true if and only if the pre-order value of `w` is in between the pre and post-order values of `v` i.e. $v_{\text{pre}} < w_{\text{pre}} < v_{\text{post}}$. Notice that $E_{\text{pre}}(= 5)$, $B_{\text{pre}}(= 1)$ and $B_{\text{post}}(= 8)$ in Figure 4.1 satisfies the condition hence E is reachable from B. However interval based indexing doesn't address the evolving nature of the graph. As the graph is constantly changing, the computed pre and post order indices becomes stale. Hence the new indices needs to be computed as the graph evolves. In order to update these stale indices efficiently we propose a novel algorithm which is discussed in Section 4.5.2.

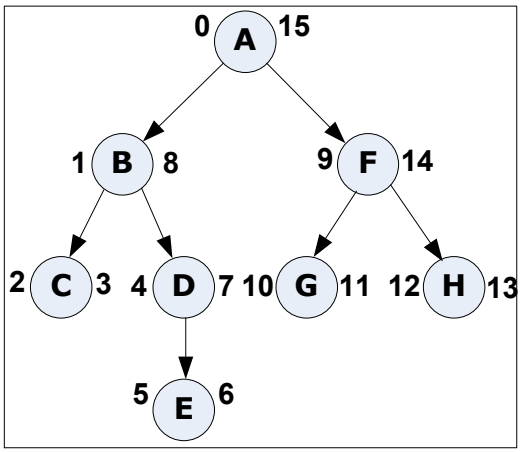

Figure 4.1: Interval Based Indexing Scheme

4.4 Naive Approach and its Drawbacks

4.4.1 Process All Queries

As mentioned earlier the graph is constantly evolving, hence a set of queries that were reachable might become unreachable or a set of queries that were unreachable might become reachable. A naive approach to identify a set of queries that have become unreachable/reachable from a given set of queries is to reevaluate every single query. We call this approach Process All Queries (PAQ). However there are drawbacks to this approach. One of the main drawbacks is computational overhead for evaluating every query for detecting the change in its state (unreachable/reachable). It is unnecessary to evaluate every query as the status of lot of queries will be unaffected because of the edits happening at the given time instance. We realize PAQ using Depth First Traversal (DFT) for reevaluating the status of queries.

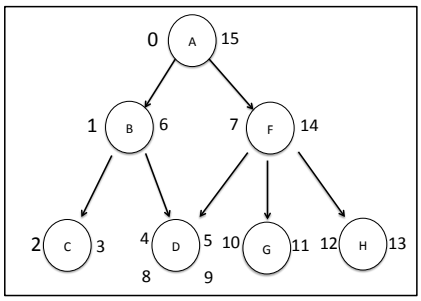

Figure 4.2: Demonstration of the Algorithm

4.5 CoUPE Approach

4.5.1 Selective Query Processing Scheme

CoUPE has two major components. First, a technique for deciding which subset of given queries are likely to get affected because of the most recent edit in the graph called Selective Query Processing Scheme (SQPS). Second, indexing the evolving graph.

In SQPS, graph is indexed using interval based indexing technique and initial state of the given queries is evaluated using these indices. As graph evolves, indices related to graph needs to be updated to maintain connectivity information and reevaluate the state of queries to check if any queries that were reachable has become unreachable and vice versa. A naive way is to evaluate reachability of every query using updated indices but its unnecessary and computationally expensive to evaluate every query. In order to overcome this problem, we have developed a technique that identifies a potential set of queries that are likely to get affected because of the most recent edit to the graph and we reevaluate the state of only these queries. We present this novel heuristic in the following algorithm.

4.5.2 Algorithm

We now outline our algorithm for continuously evaluating the queries for every structural change in the graph. First, we explain important notations that we will use in our algorithm description. $QueryList$ is the initial set of issued queries. $EvalQueryList$ is used to keep track of potential queries that needs to be reevaluated after every structural change in the graph. Algorithm 4 describes the heuristic for identifying subset of queries from $QueryList$ that are likely to become unreachable because of edge deletions in the graph. Algorithm 4 shows the pseudo-code of our algorithm. In the initialization phase of the algorithm (lines 1-2 of the algorithm), describes a query list (QueryList) that consists of all issued queries and a query list ($EvalQueryList$) for tracking potential queries for reevaluation. In line 3, we pre process the query list in which we capture the initial state(reachable or unreachable) of every query based on the current state of the graph. In Figure 4.2, let us assume $QueryList$ consists of following queries.

- $A \rightarrow D$, State: Reachable
- $B \rightarrow D$, State: Reachable
- $F \rightarrow G$, State: Reachable

Consider a query q which checks the reachability from vertex *v* to vertex *u* in the given graph. If vertex *u* is reachable from vertex *v* then we track all the edges responsible for this state using indices provided by interval based indexing technique. Let's assume vertex *u* is directly reachable from *v* then we save pre-index of v (v^q_{pre}), pre-index of u (u^q_{pre}), post-index of v (v^q_{post}) as a triple to capture the state. Let us annotate the queries in above example with triples. In this example, the following query $A \rightarrow D$ has multiple paths. We randomly choose one of the paths to annotate the queries with the indices.

- $A \rightarrow D$, State: Reachable, Indices: (0, 4, 15)

- $B \rightarrow D$, State: Reachable, Indices: (1, 4, 6)
- $F \rightarrow G$, State: Reachable, Indices: (7, 10, 14)

In line 4, an edit from the edit stream is processed to modify the structure of the graph. In line 6-12, for every edit that deletes an edge in graph we mark the portion of reachable queries that might become unreachable using the state information acquired in the pre processing step as described in line 7 and add the query to $EvalQueryList$ as shown in line 9. $EvalQueryList$ are the list of queries that needs to be reevaluated because of the edge deletion. In Figure 4.2, let us assume an edge from node B to node D is deleted. We iterate over the queries to check if any of the queries might be affected and get rid of any stale indices. In the following example query $A \rightarrow D$ and $B \rightarrow D$ are added to $EvalQueryList$ for reevaluation.

- $A \rightarrow D$, State: Unknown (*needs re-evaluation*)
- $B \rightarrow D$, State: Unknown (*needs re-evaluation*)
- $F \rightarrow G$, State: Reachable, Indices: (7, 10, 14)

Before reevaluating the queries, we need to update the indices of the graph to capture the latest connectivity information. Lines 13-20, updates the graph's indices or connectivity information. In lines 23-31, we reevaluate the queries in $EvalQueryList$ to check their state (reachable or unreachable). We trigger an alert if any reachable queries become unreachable. In the following example $A \rightarrow D$ is still reachable through a non tree edge but $B \rightarrow D$ is unreachable.

- $A \rightarrow D$, State: Reachable, Indices: (0, 8, 15)
- $B \rightarrow D$, State: Unreachable.
- $F \rightarrow G$, State: Reachable, Indices: (7, 10, 14)

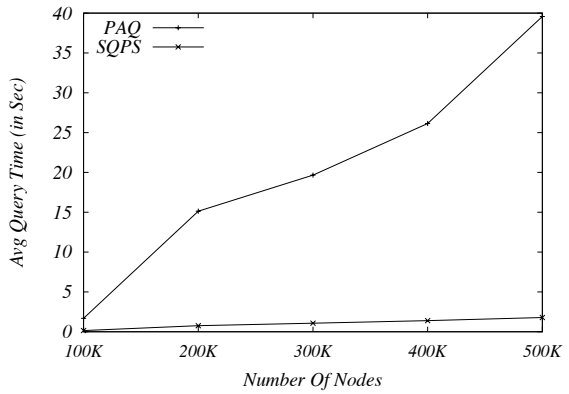

Figure 4.3: Avg Query Latency (Varying Vertices)

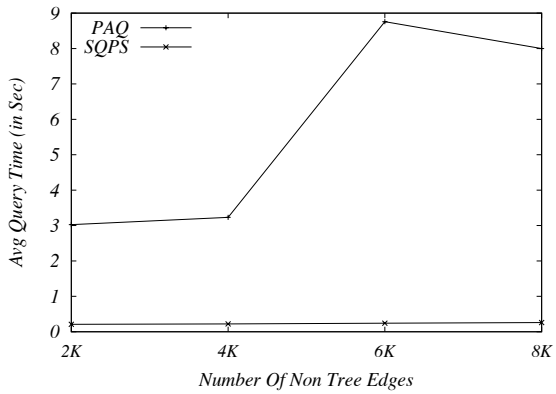

Figure 4.4: Avg Query Latency (Varying Non-Tree Edges)

4.6 Experiments and Results

We now discuss the experiments we have performed to evaluate our SQPS (Selective Query Processing Scheme). We have implemented our CoUPE framework in Java. The implementation is done in a modular fashion so that specific components can be enabled or disabled for evaluation. We compare SQPS with traditional scheme discussed in Section 4.4, namely, process all queries (PAQ). PAQ uses depth-first traversal (DFT) to evaluate reachability of a query.

We have evaluated our framework on AWS using t2.medium instances with 2.5GHz intel Xeon processors and 4GB RAM that runs 64 bit Ubuntu. MySQL server has been used for storing the pre and post-order index values.

To the best of our knowledge, there are no publicly available time evolving graph 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 scheme, we generate a number of data sets by varying important parameters (e.g., hierarchy size, number of non-tree edges, number of edits etc.). The generator program accepts hierarchy size (in terms of number of vertices), number

of non-tree edges, and number of edits as input parameters. Based on the specified hierarchy size, we generate a tree with each non-leaf vertices having approximately same number of children and we randomly choose two vertices from the graph and add a non-tree edge, this is controlled using the number of non-tree edges parameter.

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. To generate an delete edit, an edge is randomly chosen from set of existing edges and deleted. Edit lists used in our experiments have only delete edits as we are testing set of reachable queries that becomes reachable because of an *edge-delete* edit. The query workload is generated in the following manner. For each query, the source vertex and the destination vertex are all chosen randomly.

We evaluate our framework on query latency. In order to provide better insight into query processing overheads in our framework, we measure the latency incurred by varying number of total vertices in graph as well as number of non tree edges in the graph. In each case, we report the mean over all queries.

Figure 4.3 shows the average query latencies by varying vertices from 100K to 500K. X-axis indicates the number of vertices in the graph and Y-axis indicates average time taken to reevaluate the status of the queries. The results of PAQ and SQPS, may seem intuitive. As we increase the number of vertices in the graph, we can see that time taken to reevaluate the queries using PAQ is much higher because of two reasons: 1) increase in the size of the graph and 2) Reevaluating every single query is computationally expensive and wasteful. On the other hand CoQS performs significantly better because it doesn't evaluate every single query after a structural change in the graph.

Figure 4.4 shows the average query latencies by varying number of non tree edges from the

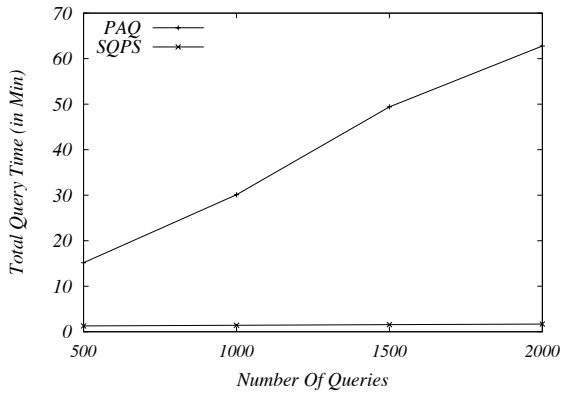

Figure 4.5: Overall Query Latency (Varying Queries)

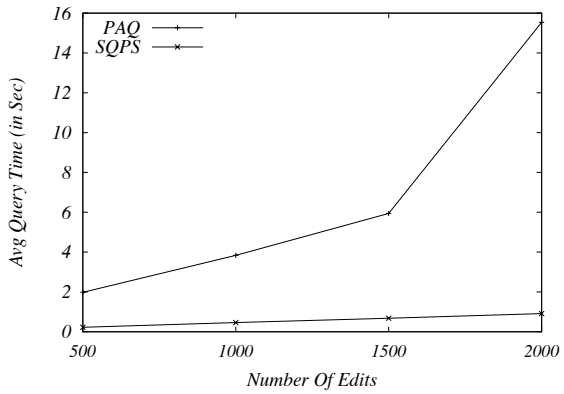

Figure 4.6: Avg Query Latency (Varying Edits)

graph. X-axis indicates the number of non-tree in the graph and Y-axis indicates average time taken to reevaluate the status of the queries. Number of nodes in the graph remained constant as we varied the number of non-tree edges. This experiment was conducted on a graph with 100K vertices. As we can see that number of non tree edges has very little effect on SQPS framework but PAQ is more sensitive to number of non tree edges as it has more paths to explore while reevaluating the queries.

Figure 4.5 shows the overall query latency by varying number of queries. X-axis represents the total number of queries and Y-axis represents the overall query latency to answer theses queries. As we increase the number of queries, overall time taken to reevaluate these queries increase in PAQ because every single query needs to be evaluated and there is a very little increase in overall query time of SQPS as expected.

Figure 4.6 shows the average query latency by varying number of edits. X-axis represents the total number of edits and Y-axis represents average query latency. As we increase the number of edits time taken to check the state of the query increases because performing an edit involves

reevaluating the queries. In PAQ, as we reevaluate all the queries, average query latency is significantly higher compared to SQPS where we reevaluate only subset of queries identified by our algorithm.

To summarize, through our experimental study we have demonstrated that the CoUPE framework significantly better at detecting the change in the state of queries for an evolving graph.

Algorithm 4 ReachableToUnReachable($QueryList$)

$QueryList \leftarrow Given\ query\ set$
$EvalQueryList \leftarrow Declare\ empty\ list$
$QueryList \leftarrow Pre-process\ (QueryList)$
for all Edit e ($s \rightarrow t$) in edit stream **do**
  $EvalQueryList \leftarrow Initialize\ empty\ list$
  for all Query q ($v \rightarrow u$) in QueryList **do**
    if $v^q_{pre} < t^e_{pre} < u^q_{pre} and\ u^q_{pre} < t^e_{post} < v^q_{post}$ **then**
      $Mark\ query\ for\ reevaluation$
      $EvalQueryList \leftarrow q$
      $Invalidate\ Connectivity\ info$
    end if
  end for
  if $e\ is\ DELETE\ edit$ **then**
    if $e\ is\ tree\ edge$ **then**
      $Remove\ tree\ edge\ related\ indices$
      $Reindex\ parts\ of\ graph\ if\ necessary$
    else if $e\ is\ non\ tree\ edge$ **then**
      $Remove\ non\ tree\ edge\ indices$
      $Update\ transitive\ closure$
    end if
  end if
end for
for all Query q in EvalQueryList **do**
  if $q\ Reachable\ via\ Tree\ Edge$ **then**
    $Update\ tree\ edge\ connectivity\ info$
  else if $q\ Reachable\ via\ Non\ Tree\ Edge$ **then**
    $Update\ non\ tree\ edge\ connectivity\ info$
  else
    $Declare\ Unreachable$
  end if
end for

Chapter 5

Related Work

Graph is a fundamental data structure used for data modeling in many domains such as social networks [59, 47, 10], evolutionary genomics [12], geographic information systems [36], world wide web [31, 1] etc. The graphs in many of these emerging domains are not only huge but also undergoes constant changes. For instance, consider the relationship graph of a social network. Importance of querying massively large graphs for understanding and extracting useful pieces of information has evoked a huge interest in the recent past among many research communities.There are several types of queries for mining interesting pieces of information in graphs.

Distance Queries [82], Reachability Queries [71, 75], Regular Path Queries [46], Shortest Path Queries [25] etc. are few important types of queries in graphs. Shortest path queries [25] tries to find a shortest path between any two given vertices in the graph. Regular Path Queries (RPQ) [46] is a regular expression R over the (edge or node) labels of a graph G. Reachability Queries [75] tries to find if there exists a path between any two given vertices in a directed graph. Distance Queries [82] tries to find distance from a given vertex in graph to any other vertex in the graph. Also, there is a different class of queries called *continuous queries [15].* These queries are are issued once and run continually over the evolving dataset. Continuous query processing systems [6, 57] are fundamentally different from conventional query processing systems. Continuous

queries on data streams [51, 9, 32], complex events [76] and databases [60, 78] has been studied extensively and has strong application in many domains such as computer vision, finance, network security, sensor networks etc. Continuous queries were used in Tapestry system [70] for content-based filtering of email and bulletin board messages. OpenCQ [53] and NiagaraCQ [17] systems used continuous queries for monitoring persistent data sets spread over a wide-area network.

Continuous query processing in evolving graphs have entirely new dimension to the problem i.e. every query is continually ran to eternity (in general), hence the query set is also continuously evolving. Exhaustively researched field of continuous query systems related to data streams/databases will help us identify important semantics, constraints and architectural features that are also applicable in real-time graph analytics [21] and continuous query processing in evolving graphs . A new generation continuous query processing system for evolving graphs should not only consider the scale of the data but also the sheer complexity of the different types of queries.

5.1 Graph Mining

Graph mining has been an active research area. Computing communities, spectral clustering, diameter estimation, connected components etc. are few popular graph mining operations. There are massive number of algorithms that have been proposed in the literature for supporting these graph mining operations, computing communities [18, 28, 42], subgraph discovery [43, 37, 37, 83, 16, 79], finding important nodes [13, 45], computing number of triangles [72, 73], connected components [65, 8, 38] etc. In the past most of these mining algorithms made an assumption that the complete graph fits in memory, or a single disk. Hence these algorithms doesn't scale well, at lease directly as the size of the graphs reaches to several million nodes.

5.2 Graph Analytics

In the recent past there has been huge interest in real-time graph analytics [58, 33, 77] because of the exponential growth in streaming data from social networks. Numerous systems like Pregel [58], GraphLab [54], GraphX [77] etc have been proposed to run different types of queries like distance queries, shortest path queries, reachability queries etc for graph analytics. on large graphs. Pregel and GraphLab are based on graph-parallel computation where GraphLab unifies graph-parallel and data-parallel computation. Graph-parallel computation uses a vertex centric view of graphs similar to data-parallel systems (MapReduce [24] and Spark [81]) using record centric view of collections but graph-parallel computation derives parallelism by partitioning the graph data across processing resources where as data-parallel computation derives parallelism by processing independent data on processing resources. Using data-parallel system like MapReduce [24] and Spark [81] for graph processing is challenging as it causes excessive data movement across processing resources due to the failure to exploit graph structure. Hence in order to create a graph analytics pipeline, composing graph-parallel systems for graph processing and data-parallel systems for graph loading resulted in a complex abstraction. GraphX presents a unified abstraction by combining graph-parallel and data-parallel computations into a single system. Also, there are several high performance graph processing libraries like Pegasus [41], ScaleGraph [11], Parallel Boost Graph Library [34], KDT [56], the Combinatorial BLAS [14] etc. that are helpful for large scale graph processing.

5.3 Reachability Analysis

Efficient processing of reachability queries for static trees and more generally for static graphs has been an active topic of research [35, 64, 80, 3, 22]. Many of the recent works on reachability indexes for trees and directed acyclic graphs have been done in the context of XML query processing. One of the simplest mechanisms that can be employed to answer reachability queries on graphs is

to traverse the graphs using breadth first or depth first approach during query time [23]. Another approach is to pre compute transitive closure(TC) [4, 55] of a graph where transitive closure consists of set of node pairs (p, q) for which a path exists from p to q in the given graph. But the size of TC is $O(n^2)$ and its computation cost is $O(n^3)$. Hence both these approaches (Transitive Closure and Traversal) are not applicable for large graphs.

In order to address these problems several indexing techniques have been proposed by researchers over time. As mentioned before, interval-based indexing is a prominent approach in this regard [35]. The other approaches include 2-hop cover [22], chain decomposition [39] etc. 2-hop cover is a collection of shortest paths or full paths in a graph such that for every pair of vertices (u, v), there is a path from u to v that is a concatenation of two paths from the collection if a path exists. 2-hop cover requires $O(nm^{1/2})$ space. However the problem of computing a 2-hop cover is NP-hard [64] stands for 2-HOP cover Index. It is a connection index for XML collections that is based on 2-hop cover, which reduces the labeling complexity of original 2-Hop cover algorithm to $O(n^3)$. This technique performs well on forests with fewer connections between different sub-trees but doesn't scale well for denser graphs. Agrawal et al. [3] proposed a strategy similar to interval-based indexing for answering reachability queries in DAGs. It assigns multiple non-overlapping intervals to each vertex; reachability testing from vertex v to vertex w is done by checking whether every interval of w is contained by some interval of v.

GRIPP [71], DualLabeling [75] and GRAIL are 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 initially assigns interval-based indexing labels to the nodes in spanning tree of a given graph and keeps track of non-tree edges in a transitive link table (TLT) where the number of non-tree edges in TLT depends on the choice of the spanning tree. Similarly, GRIPP assigns at least a pair of pre and postorder labels to every node in the graph. It also assigns an additional pair of labels to a node when it has multiple parents. The additional pair of labels assigned to a node encodes non-

tree edges in the graph. Hence some nodes will get more than a pair of pre and postorder values. GRAIL on the other hand assigns multiple intervals to every node in the graph via random graph traversals which is another variant of interval based indexing. The key idea here is to eliminate pairs of query nodes that are not reachable using the intervals assigned to every node. Most of these works are not designed for dynamic trees/graphs. While Schekel et al. discuss incremental maintenance of the HOPI index [64], 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.

In general, research on indexing and querying TEGs is extremely limited [63, 19]. Shirani-Mehr et al. [66] proposed two indexes called ReachGrid and ReachGraph for evaluation of reachability queries in spatiotemporal contact datasets. Ren et al. [63] study shortest path queries in TEGs, while Wang et al. [74] propose a scheme for continuous sub-graph queries. On the other hand, mining TEGs has been an active area of research [68, 30, 29, 26, 44]. While some of these works propose to use TEG queries, they do not provide query techniques. Several empirical studies have been performed on TEGs in various domains [50, 49, 48, 2, 27, 52, 69, 7, 67].

Chapter 6

Conclusion

In the recent years, graphs in many emerging domains are not only massive but also constantly evolving. Graph analytics on these big graphs has become a prominent research area in the field of computer science. Several distributed graph processing models (Pregel, GraphX), platforms (Apache-Giraph) and libraries (Pegasus, ScaleGraph) are implemented for running graph analytics and handling the sheer scale of today's graphs as the old graph processing paradigms are not efficient in handling the scale and evolving nature of graph in the recent times. There are different types of queries that can be executed on these massive graphs for extracting interesting pieces of information. One of the important types of queries is reachability queries.

Efficient and scalable processing of reachability queries in evolving hierarchies is important for many modern applications. In this research, we presented a tunable, time and space efficient framework called SCISSOR (***s**elec**t**ive **s**naps**h**o**t** indexing with progressive solution **r**efinement*) for testing reachability between given pair of vertices on any given snapshot of a evolving hierarchy. The main idea behind SCISSOR is to selectively index a subset of snapshots of a evolving hierarchy and use these snapshot indices to answer reachability queries on all snapshots of the evolving hierarchy. Our framework includes three novel features – (1) an efficient algorithm for analyzing the effect of the changes that have occurred between snapshots on the reachability from one given

vertex to another; (2) an adaptation of the interval-based indexing strategy for evolving hierarchies, called, non-contiguous interval index (NCI index); and (3) a heuristic-based technique for deciding which snapshots to index.

Furthermore, we leveraged our novel framework for answering version-specific XPath expressions in Continuously evolving XML (CEXML) repositories. CEXML documents are employed for representing dynamic information in many emerging domains such as employee information systems and geographical information systems. In such domains, it is often important to evaluate XPath expressions on certain specific version of a CEXML document. As a part of this research, we introduced the concept of version-specific XPath expression for modeling such queries. We presented a tunable, time and space efficient framework for evaluating version-specific XPath expressions on CEXML documents.

In this research, we also address the problem of answering continuous reachability queries where queries are issued only once but run "continually" over the evolving graph. Scalable processing of such queries in evolving graphs is important in many modern domains such as online social networks and overlay network connectivity analysis. We present an efficient and scalable framework called CoUPE (**Co**ntinuous q**U**ery **P**rocessing **E**ngine) for answering continuous reachability queries in evolving graphs as a part of this research. The main idea here is to index the evolving graph and compute the status of only the subset of queries that are likely to be affected by each incoming edit happening on the dynamic graph. The CoUPE framework includes two novel features, namely, an algorithm to recompute the indices of evolving graph efficiently and a scalable heuristic-based technique to identify the subset of queries that are likely to get impacted by an incoming edit.

Bibliography

[1] L. Adamic. World wide web, graph structure. *Encyclopedia of Complexity and Systems Science*, pages 10058–10072, 2009.

[2] C. C. Aggarwal and P. S. Yu. Online analysis of community evolution in data streams. In *SDM*, 2005.

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

[4] R. Agrawal and H. V. Jagadish. Direct algorithms for computing the transitive closure of database relations. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB)*, pages 255–266, 1987.

[5] H. Aït-Kaci, R. S. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(1), 1989.

[6] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004.

[7] S. Asur, S. Parthasarathy, and D. Ucar. An event-based framework for characterizing the evolutionary behavior of interaction graphs. In *KDD*, 2007.

[8] B. Awerbuch and T. Singh. New connectivity and msf algorithms for ultracomputer and pram. In *ICPP*, volume 83, pages 175–179, 1983.

[9] S. Babu and J. Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120, 2001.

[10] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 44–54, New York, NY, USA, 2006. ACM.

[11] N. T. Bao and T. Suzumura. Towards highly scalable pregel-based graph processing platform with x10. In *Proceedings of the 22nd international conference on World Wide Web companion*, pages 501–508. International World Wide Web Conferences Steering Committee, 2013.

[12] M. H. Birte Kehr, Kathrin Trappe and K. Reinert. Genome alignment with graph data structures: a comparison. *BMC Bioinformatics*, 2014.

[13] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1):107–117, 1998.

[14] S. Brohee and J. Van Helden. Evaluation of clustering algorithms for protein-protein interaction networks. *BMC bioinformatics*, 7(1):488, 2006.

[15] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668. ACM, 2003.

[16] C. Chen, X. Yan, F. Zhu, and J. Han. gapprox: Mining frequent approximate patterns from a massive network. In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, pages 445–450. IEEE, 2007.

[17] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 379–390, New York, NY, USA, 2000. ACM.

[18] J. Chen, O. R. Zaïane, and R. Goebel. Detecting communities in social networks using max-min modularity. In *SDM*, volume 3, pages 20–24. SIAM, 2009.

[19] L. Chen and C. Wang. Continuous Subgraph Pattern Search over Certain and Uncertain Graph Streams. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(8), 2010.

[20] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computing reachability labelings for large graphs with high compression rate. In *EDBT*, 2008.

[21] S. Choudhury, L. Holder, G. Chin, and J. Feo. Large-scale continuous subgraph queries on streams. In *Proceedings of the First Annual Workshop on High Performance Computing Meets Databases*, HPCDB '11, pages 29–32, New York, NY, USA, 2011. ACM.

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

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[24] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[25] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[26] D. M. Dunlavy, T. G. Kolda, and E. Acar. Temporal link prediction using matrix and tensor factorizations. *Transactions on Knowledge Discovery from Data (TKDD)*, 5(2), 2011.

[27] T. Falkowski, J. Bartelheimer, and M. Spiliopoulou. Mining and visualizing the evolution of subgroups in social networks. In *Web Intelligence*, 2006.

[28] T. Falkowski, A. Barth, and M. Spiliopoulou. Dengraph: A density-based community detection algorithm. In *Web Intelligence, IEEE/WIC/ACM International Conference on*, pages 112–115. IEEE, 2007.

[29] C. Faloutsos, T. G. Kolda, and J. Sun. Mining large graphs and streams using matrix and tensor tools. In *SIGMOD Conference*, page 1174, 2007.

[30] J. Ferlez, C. Faloutsos, J. Leskovec, D. Mladenic, and M. Grobelnik. Monitoring network evolution using mdl. In *ICDE*, 2008.

[31] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a web-site management system. *SIGMOD Record*, 26:4–11, 1997.

[32] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 500–511. VLDB Endowment, 2003.

[33] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[34] D. Gregor and A. Lumsdaine. The parallel bgl: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, 2:1–18, 2005.

[35] T. Grust, M. van Keulen, and J. Teubner. Accelerating xpath evaluation in any rdbms. *Transactions on Database Systems (TODS)*, 29, 2004.

[36] R. H. Güting. Graphdb: Modeling and querying graphs in databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 297–308, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[37] P. Hintsanen and H. Toivonen. Finding reliable subgraphs from large probabilistic graphs. *Data Mining and Knowledge Discovery*, 17(1):3–23, 2008.

[38] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

[39] H. V. Jagadish. A compression technique to materialize transitive closure. In *ACM Transactions on Database Systems (TODS)*, pages 558–598, 1990.

[40] R. Jin, L. Liu, B. Ding, and H. Wang. Distance-Constraint Reachability Computation in Uncertain Graphs. *The Proceedings of the VLDB Endowment (PVLDB)*, 4(9), 2011.

[41] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 229–238. IEEE, 2009.

[42] G. Karypis and V. Kumar. Parallel multilevel series k-way partitioning scheme for irregular graphs. *Siam Review*, 41(2):278–300, 1999.

[43] Y. Ke, J. Cheng, and J. X. Yu. Top-k correlative graph mining. In *SDM*, volume 2, pages 150–163. SIAM, 2009.

[44] M.-S. Kim and J. Han. A particle-and-density based evolutionary clustering method for dynamic networks. *The Proceedings of the VLDB Endowment (PVLDB)*, 2(1), 2009.

[45] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5):604–632, 1999.

[46] A. Koschmieder and U. Leser. Regular path queries on large graphs. In *Proceedings of the 24th International Conference on Scientific and Statistical Database Management*, SSDBM'12, pages 177–194, Berlin, Heidelberg, 2012. Springer-Verlag.

[47] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media?

[48] J. Leskovec, L. Backstrom, R. Kumar, and A. Tomkins. Microscopic evolution of social networks. In *KDD*, 2008.

[49] J. Leskovec and E. Horvitz. Planetary-scale views on a large instant-messaging network. In *WWW*, 2008.

[50] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *Transactions on Knowledge Discovery from Data (TKDD)*, 1(1), 2007.

[51] H.-S. Lim, J.-G. Lee, M.-J. Lee, K.-Y. Whang, and I.-Y. Song. Continuous query processing in data streams using duality of data and queries. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 313–324. ACM, 2006.

[52] Y.-R. Lin, Y. Chi, S. Zhu, H. Sundaram, and B. L. Tseng. Facetnet: a framework for analyzing communities and their evolutions in dynamic networks. In *WWW*, 2008.

[53] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. on Knowl. and Data Eng.*, 11(4):610–628, July 1999.

[54] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[55] H. Lu. New strategies for computing the transitive closure of a database relation. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB)*, pages 267–274, 1987.

[56] A. Lugowski, D. M. Alber, A. Buluç, J. R. Gilbert, S. P. Reinhardt, Y. Teng, and A. Waranis. A flexible open-source toolbox for scalable complex graph analysis. In *SDM*, volume 12, pages 930–941. SIAM, 2012.

[57] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 49–60, New York, NY, USA, 2002. ACM.

[58] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[59] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *In Proceedings of the 5th ACM/USENIX Internet Measurement Conference (IMC07*, 2007.

[60] M. F. Mokbel, X. Xiong, and W. G. Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 623–634, New York, NY, USA, 2004. ACM.

[61] P. R. Mullangi, G. Penematsa, and L. Ramaswamy. Scalable xpath evaluation on large-scale continuously evolving xml repositories. In *Proceedings of the 2014 IEEE International Congress on Big Data*, BIGDATACONGRESS '14, 2014.

[62] P. R. Mullangi and L. Ramaswamy. Scissor: Scalable and efficient reachability query processing in time-evolving hierarchies. In *Proceedings of the 22Nd ACM International Conference on Conference on Information & Knowledge Management*, CIKM '13, 2013.

[63] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On Querying Historical Evolving Graph Sequences. *The Proceedings of the VLDB Endowment (PVLDB)*, 2011.

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

[65] Y. Shiloach and U. Vishkin. An o (logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[66] H. Shirani-Mehr, F. B. Kashani, and C. Shahabi. Efficient reachability query evaluation in large spatiotemporal contact datasets. *CoRR*, abs/1205.6696, 2012.

[67] B. Sun, G. Shu, A. Podgurski, S. Li, S. Zhang, and J. Yang. Propagating bug fixes with fast subgraph matching. In *ISSRE*, 2010.

[68] J. Sun, C. Faloutsos, S. Papadimitriou, and P. S. Yu. Graphscope: parameter-free mining of large time-evolving graphs. In *KDD*, 2007.

[69] L. Tang, H. Liu, J. Zhang, and Z. Nazeri. Community evolution in dynamic multi-mode networks. In *KDD*, 2008.

[70] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, SIGMOD '92, pages 321–330, New York, NY, USA, 1992. ACM.

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

[72] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 837–846. ACM, 2009.

[73] C. E. Tsourakakis, M. N. Kolountzakis, and G. L. Miller. Approximate triangle counting. *arXiv preprint arXiv:0904.3761*, 2009.

[74] C. Wang and L. Chen. Continuous subgraph pattern search over graph streams. In *ICDE*, 2009.

[75] 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.

[76] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418. ACM, 2006.

[77] R. S. Xin, D. Crankshaw, A. Dave, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: Unifying data-parallel and graph-parallel analytics. *CoRR*, abs/1402.2394, 2014.

[78] X. Xiong, M. F. Mokbel, and W. G. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 643–654. IEEE, 2005.

[79] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 721–724. IEEE, 2002.

[80] H. Yildirim, V. Chaoji, and M. J. Zaki. GRAIL: Scalable Reachability Index for Large Graphs. *The Proceedings of the VLDB Endowment (PVLDB)*, 3(1), 2010.

[81] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[82] A. D. Zhu, X. Xiao, S. Wang, and W. Lin. Efficient single-source shortest path and distance queries on large graphs. *CoRR*, abs/1306.1153, 2013.

[83] F. Zhu, X. Yan, J. Han, and S. Y. Philip. gprune: a constraint pushing framework for graph pattern mining. In *Advances in Knowledge Discovery and Data Mining*, pages 388–400. Springer, 2007.