Aristotle University of Thessaloniki

Doctoral Thesis

Management of Historical Information and Preference in Massive Datasets

Author: Andreas Kosmatopoulos

Supervisor: Prof. Kostas Tsichlas

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy in the

Data and Web Science Laboratory
School of Informatics

June 30, 2020
Dedicated to the memory of my father
Declaration of Authorship

I, Andreas KOSMATOPOULOS, declare that this thesis titled, ”Management of Historical Information and Preference in Massive Datasets“ and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.

- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.

- Where I have consulted the published work of others, this is always clearly attributed.

- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.

- I have acknowledged all main sources of help.

Signed: ____________________________________________

Date: ____________________________________________

Doctoral Committee:

- Assistant Professor Kostas Tsichlas
- Associate Professor Apostolos N. Papadopoulos
- Professor Athena Vakali
- Assistant Professor Grigoris Tsoumakas
- Associate Professor Loukas Georgiadis
- Associate Professor Christos Makris
- Professor Nikolaos Samaras
Abstract

Faculty of Sciences
School of Informatics

Doctor of Philosophy

Management of Historical Information and Preference in Massive Datasets

by Andreas Kosmatopoulos

A distinctive phenomenon of the past two decades has been the explosive data generation increase that is witnessed through a multitude of sources. Social networks and content provider websites as well as organizations and enterprises in other disciplines such as astronomy and genomics report daily data storage and data processing quantities of demanding volume. It is important, therefore, for solutions that target problems in this modern environment to be largely space efficient while retaining their time efficiency. To this end, this thesis tackles challenges related to preference query evaluation as well as historical information management by proposing algorithms and data structures that are linear in their space cost. More specifically, the first part of the thesis deals with the dynamic maintenance of top-k dominating points and the evaluation of dynamic 3-sided skyline queries in the presence of dataset updates. The second part of the thesis presents a prototype system termed HiNode that maintains graph historical information from evolving graph sequences in an asymptotically space-optimal manner. Finally, two implementations of HiNode with different underlying technologies are described in detail and are extensively evaluated.
Acknowledgements

During the course of this journey I’ve had the luck and privilege of meeting and associating with a number of exceptional people that assisted me in completing my goals. First and foremost, I would like to express my sincere gratitude to my advisor Assistant Professor Kostas Tsichlas for his support through all these years and for being the primary source of academic inspiration that has enabled me to pursue and complete my doctoral studies. His passionate and engaging lecturing and problem-solving approaches were the prime factors for my initial interest in studying algorithm and data structure design and played a key role into my overall development and progress as an academic researcher.

Next, I am extremely grateful for the assistance Assistant Prof. Anastasios Gounaris provided during my studies. Apart from being a vigorous collaborator in several manuscripts and technical projects, Prof. Gounaris went out of his way numerous times to aid me and ensure that I successfully completed my studies even through hardships.

Moreover, I am grateful to Associate Prof. Apostolos N. Papadopoulos as well as Prof. Athena Vakali for their lasting encouragement and continuing trust throughout my endeavor while being members of my PhD advisory board. I would also like to gratefully acknowledge Assistant Prof. Grigoris Tsoumakas (Aristotle University of Thessaloniki), Associate Prof. Loukas Georgiadis (University of Ioannina), Associate Prof. Christos Makris (University of Patras) and Prof. Nikolaos Samaras (University of Macedonia) for their time and participation as members of the dissertation examination committee. Finally, I wish to thank Prof. Yannis Manolopoulos for initially welcoming me as a member of the DataLab (then DeLab) research group.

Heartfelt thanks go out to my fellow lab members Dr. Georgia Kougka, Dr. Athanasios Naskos and Dr. Antonia Gogoglou who were always there to assist me during tough times as well as celebrate with me the happy ones. Additionally, I am extremely grateful to Christos Bellas and Theodoros Toliopoulos for our excellent collaboration and for their assistance in various technical and implementation details in my PhD thesis. Another person I wish to sincerely thank is Dr. Christos Zigkolis for all the insightful discussions we had during my first year as a PhD candidate. Last but not least, I would like to thank all the lab members I’ve met in my last year of doctoral studies for creating a friendly environment, and especially Valentina Michailidou, Maria Pegia, Georgia Baltou, Nikodimos Nikolaidis and Ioannis Christoforidis.

I want to thank my good friends Eleni, Eirini and Alexandros for all the fun times we had and for their encouragement at one point or another through these years. I will forever be indebted to a close group of friends with whom I shared many beautiful experiences that helped me discover new and interesting sides of myself. Panos, Olga, Petros, Ioanna and Stelios
all had a significant part in the completion of this PhD thesis and for this I am truly grateful.

Last but not least, I want to thank my family which was one of the biggest contributors to the completion of my doctoral studies. My brother, Michalis, had always been encouragingly and supportively on my side ever since the start of my studies willing to help with even the most obscure technical issue that occurred and for that I deeply thank him. Words cannot express the immense support that my mother Evi selflessly provided in all aspects of my life. I am extremely grateful for her tirelessness and loving compassion she offered when I confronted her with any frustrating issue that I had and I am truly blessed to have her in my life.

Finally, I want to express my gratitude to my late father Kostas Kosmatopoulos who had been a Lecturer at the Physics Department of Aristotle University of Thessaloniki up until he passed away in 2009. I wholeheartedly thank him for his invaluable knowledge, his kindness and his unending patience in assisting me when I was confronted with a problem. He had been a person loved not only by his family but by his academic and social colleagues and students as well. I wish he were here to see me complete my studies and I hope he would be pleased by the contents of this thesis.
# Contents

**Declaration of Authorship** v  

**Abstract** vii  

**Acknowledgements** ix  

## 1 Introduction 1  
1.1 Preference Queries 2  
1.2 Historical Information Management 5  
1.3 Motivation and Contributions 7  
1.4 Thesis Structure 9  

## 2 Background and Literature Review 11  
2.1 Preference Queries 11  
2.1.1 Skyline Queries 11  
2.1.2 Top-k Dominating Queries 13  
2.2 Historical Information Management 15  
2.2.1 Centralized Methods 15  
2.2.1.1 The FVF Framework 16  
2.2.1.2 Using Graph Deltas for Historical Queries 17  
2.2.1.3 Compact Sequence Representations 17  
2.2.1.4 Constructing Indices for Specific Queries 19  
2.2.2 Parallel and Distributed Methods 20  
2.2.2.1 The DeltaGraph and Historical Graph Store Systems 20  
2.2.2.2 The G* Graph Database 21  
2.2.2.3 The GreyCat system 21  

## 3 Dynamic Top-k Dominating Queries 23  
3.1 Preliminaries 24  
3.1.1 Two-dimensional Layers of Minima 25  
3.1.2 Reporting Lemma 25  
3.2 The Semi-Dynamic Case 26  
3.2.1 The Augmented \((a, b)\)-Tree 27  
3.2.1.1 Operations 29  
3.2.2 Insertion 31  
3.2.3 Query 35  
3.2.4 Reducing the Update Cost 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3 The Fully-Dynamic Case</td>
<td>36</td>
</tr>
<tr>
<td>3.4 Results for Word-RAM</td>
<td>38</td>
</tr>
<tr>
<td>4 Dynamic 3-sided Skyline Queries</td>
<td>41</td>
</tr>
<tr>
<td>4.1 Fundamental Concepts</td>
<td>42</td>
</tr>
<tr>
<td>4.1.1 Probability Distributions</td>
<td>43</td>
</tr>
<tr>
<td>4.1.2 Auxiliary Data Structures</td>
<td>43</td>
</tr>
<tr>
<td>4.2 The ML-Tree</td>
<td>44</td>
</tr>
<tr>
<td>4.2.1 The Static Non-Linear-Space ML-tree</td>
<td>44</td>
</tr>
<tr>
<td>4.2.2 The Main Memory Static Linear-Space MLR-tree</td>
<td>46</td>
</tr>
<tr>
<td>4.3 The Dynamic MLR-tree</td>
<td>48</td>
</tr>
<tr>
<td>4.3.1 Exploiting the Distribution of the Elements</td>
<td>50</td>
</tr>
<tr>
<td>5 HiNode: An Asymptotically Space-Optimal Storage Model for Historical Queries on Graphs</td>
<td>53</td>
</tr>
<tr>
<td>5.1 Preliminaries</td>
<td>54</td>
</tr>
<tr>
<td>5.2 Storage Model</td>
<td>55</td>
</tr>
<tr>
<td>5.2.1 Data Structure Overview</td>
<td>56</td>
</tr>
<tr>
<td>5.2.2 Basic Operations</td>
<td>58</td>
</tr>
<tr>
<td>5.2.3 Analysis of Space and Time Complexities</td>
<td>61</td>
</tr>
<tr>
<td>5.2.4 Qualitative Comparison</td>
<td>63</td>
</tr>
<tr>
<td>5.3 Query Processing</td>
<td>67</td>
</tr>
<tr>
<td>5.3.1 Core Algorithms for Global Queries</td>
<td>68</td>
</tr>
<tr>
<td>5.3.2 Graph Sampling: A local query case-study</td>
<td>71</td>
</tr>
<tr>
<td>5.3.3 Practical Considerations</td>
<td>74</td>
</tr>
<tr>
<td>5.4 Experimental Evaluation</td>
<td>75</td>
</tr>
<tr>
<td>5.4.1 Implementation Details</td>
<td>75</td>
</tr>
<tr>
<td>5.4.2 Dataset Description</td>
<td>76</td>
</tr>
<tr>
<td>5.4.3 Snapshot Retrieval</td>
<td>77</td>
</tr>
<tr>
<td>5.4.4 Local Query Evaluation</td>
<td>77</td>
</tr>
<tr>
<td>5.4.5 Space Consumption</td>
<td>79</td>
</tr>
<tr>
<td>5.4.6 Time Efficiency for Global Queries</td>
<td>79</td>
</tr>
<tr>
<td>5.5 Discussion</td>
<td>85</td>
</tr>
<tr>
<td>6 HiNode: A Vertex-centric Modelling Approach to Maintaining Historical Graph Data</td>
<td>87</td>
</tr>
<tr>
<td>6.1 Motivation</td>
<td>88</td>
</tr>
<tr>
<td>6.2 Modelling</td>
<td>90</td>
</tr>
<tr>
<td>6.2.1 Single Table Model</td>
<td>90</td>
</tr>
<tr>
<td>6.2.2 Multiple Tables Model</td>
<td>91</td>
</tr>
<tr>
<td>6.2.3 Querying Modes</td>
<td>93</td>
</tr>
<tr>
<td>6.3 Experimental Evaluation</td>
<td>95</td>
</tr>
<tr>
<td>6.3.1 Query Definitions and Dataset Description</td>
<td>96</td>
</tr>
<tr>
<td>6.3.2 A Baseline Approach</td>
<td>98</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 The tablet computer database. ........................................... 3  
1.2 An Evolving Graph Sequence. ........................................ 6  
2.1 \( G \cup \) and \( G \cap \) for the evolving graph sequence in Figure 1.2. ... 16  
2.2 EdgeLog and EveLog for Figure 1.2. ................................. 18  
3.1 The first layer of minima (skyline). ................................. 25  
3.2 An \((a,b)\)-tree node \((k = 2)\). ....................................... 28  
3.3 Insertion of a new point \(p\) ........................................... 32  
3.4 Example of layer restructuring. ....................................... 33  
3.5 Deletion of existing points. ............................................ 37  
4.1 Example of a skyline result in a 3-sided query rectangle. ....... 42  
5.1 Evolving Directed Graph Sequence ................................. 55  
5.2 A bird’s-eye view for an instance of our proposed data structure. 58  
5.3 The data structure built on the graph sequence of Figure 5.1 . 59  
5.4 Cases of existing intervals for the field \(f\) ............................ 60  
5.5 Two-hop neighborhood query time. ................................ 78  
5.6 Vertex sampling query time .............................................. 78  
5.7 Space consumption with varying insertion to update ratio ....... 80  
5.8 Space consumption with varying granularity ........................ 81  
5.9 Time efficiency with varying insertion to update ratio ......... 83  
5.10 Time efficiency with varying granularity .......................... 84  
5.11 Time efficiency in the presence of updates ....................... 84  
5.12 Time efficiency with varying graph density ...................... 84  
6.1 Example graph sequence ............................................. 88  
6.2 Comparison of HiNode-G\(^*\) and HiNode-NoSQL ................. 90  
6.3 Local queries on the datasets of Table 6.4 ......................... 99  
6.4 Global queries on the datasets of Table 6.4 ....................... 101  
6.5 Performance ratio on local queries (Table 6.4) .................... 103  
6.6 Performance ratio on global queries (Table 6.4) .................. 104  
6.7 Querying Modes Execution Times .................................. 106
List of Tables

2.1 Summary of historical information management approaches. 22
3.1 Chapter 3 contributions. 24
5.1 Chapter 5 Notation Table 56
5.2 HiNode Qualitative Comparison 64
5.3 Description of Real Datasets 76
5.4 Snapshot(s) portion retrieval time 77
5.5 Space consumption in real datasets 79
5.6 Time efficiency in real datasets 82
6.1 Contents of ST for the graph sequence of Figure 6.1 92
6.2 Contents of MT for the graph sequence of Figure 6.1 94
6.3 Query Definitions 97
6.4 Evaluation datasets 97
6.5 Neo4j Time Cost Comparison 105
6.6 Space Utilization 106
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASKAP</td>
<td>Australian Square Kilometre Array Pathfinder</td>
</tr>
<tr>
<td>BBS</td>
<td>Branch-and-Bound-Skyline</td>
</tr>
<tr>
<td>BNL</td>
<td>Block-Nested-Loop</td>
</tr>
<tr>
<td>CGI</td>
<td>Compact Graph Index</td>
</tr>
<tr>
<td>D&amp;C</td>
<td>Divide &amp; Conquer</td>
</tr>
<tr>
<td>FVF</td>
<td>Find-Verify-Fix</td>
</tr>
<tr>
<td>HGS</td>
<td>Historical Graph Store</td>
</tr>
<tr>
<td>RMQ</td>
<td>Range Maximum Query</td>
</tr>
<tr>
<td>IDC</td>
<td>International Data Corporation</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>LCA</td>
<td>Lowest Common Ancestor</td>
</tr>
<tr>
<td>LESS</td>
<td>Linear-Elimination-Sort-for-Skyline</td>
</tr>
<tr>
<td>MLR</td>
<td>Modified Layered Range</td>
</tr>
<tr>
<td>MWG</td>
<td>Many-Worlds Graphs</td>
</tr>
<tr>
<td>NN</td>
<td>Nearest Neighbor</td>
</tr>
<tr>
<td>SFS</td>
<td>Sort-First-Skyline</td>
</tr>
<tr>
<td>TAF</td>
<td>Temporal Graph Analysis Framework</td>
</tr>
<tr>
<td>TGI</td>
<td>Temporal Graph Index</td>
</tr>
<tr>
<td>W.H.P</td>
<td>With High Probability</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

We are in the midst of a digital information era characterised by a rapid data growth. Data is created, generated and transformed at an astonishing daily rate in various structured, unstructured or semi-structured forms and by a diverse multitude of organisations, companies and institutions. For instance, video-sharing website Youtube\(^1\) sees 400 hours of content uploaded every minute [111] while social network Facebook\(^2\) generates 4 petabytes of data per day [24]. Furthermore, in the social networking platform Twitter\(^3\) there are about 6000 tweets created per second or, equivalently, 500 million tweets created per day [108].

The staggering growth of data can also be witnessed in other disciplines apart from multimedia and social network services [117]. For example, physical everyday devices such as smartwatches, GPS devices, PDAs, digital cameras, refrigerators as well as devices predominantly used in the fields of smart-cities [52], health care [58], agriculture [35] and others; generate data and form a “smart” heterogenous interconnected world-wide network referred to as the Internet of Things (IoT). The number of interconnected IoT devices is estimated to reach 30 billion [89, 84] by 2020 which represents a 33-fold increase from the 0.9 billion devices that existed in 2009 [43].

In the discipline of astronomy, the Australian Square Kilometre Array Pathfinder (ASKAP) generates 7.5 terabytes per second in sample image data format and is projected to reach up to 750 terabytes per second by 2025 [86, 106]. Meanwhile, the genomics branch of biology exhibits a comparable data growth pattern with estimates suggesting that between 100 million and 2 billion human genomes will be sequenced by 2025 resulting in a 2 to 40 exabytes space needed for their storage [109]. Overall, according to the International Data Corporation (IDC) the amount of worldwide data is predicted to reach 175 zettabytes in 2025 up from the 33 zettabytes in 2018 [99].

The aforementioned real world examples are indicative of the global explosive data growth and are all characterised by two discerning properties; massive volume and high dynamicity of data. The former property brings

\(^1\)https://www.youtube.com
\(^2\)https://www.facebook.com
\(^3\)https://www.twitter.com
forth the need for algorithms and data structures that attempt to minimize the total storage footprint and produce overall space-efficient solutions for significant problems related to data analytics. The latter property implies that the respective datasets scarcely remain the same over arbitrary periods of time and thus any developed solutions should handle insertion of new data and deletion of existing data efficiently. Building on that, and depending on the application at hand, it may also be beneficial to maintain historical information in relation to the dataset instances prior to a data update so as to provide insight regarding the nature and evolution of the dataset as time progresses.

This thesis tackles the two challenges by designing time and space efficient solutions for the problems of preference query evaluation, and historical information storage and management. More specifically, preference queries provide an intuitive way of selecting the most important objects\(^4\) in a dataset and their efficient evaluation forms the first part of the thesis. The second part of the thesis is concerned with the effective storage, management and query evaluation in historical graph data structures, i.e. data structures that retain the previous states of a graph following an arbitrary number of update operations. Data analysis tasks that consider (a part of) the entire history of a dataset are able to support more elaborate objectives such as the discovery of certain patterns or trends that occur in highly dynamic datasets as well as the study of a dataset’s inherent vertex and edge characteristics. The concepts of preference queries and historical information management are described in more detail in the following two sections.

### 1.1 Preference Queries

A compelling challenge that arises due to the sheer volume of available information in datasets today is the discernment of data objects that are “more interesting” than the rest according to an implied or user-provided preference ranking. These data objects are generally constituted by a number of conflicting attributes, in the sense that the most desirable objects tend to maximize a selection of attributes (e.g. quality of a car) while also minimizing some others (e.g. its price) at the same time. Thus, the overall problem of grouping and ranking particular data objects becomes a non-trivial task and may provide valuable insight into the most interesting members of a dataset.

The above points will be illustrated through a running example. Consider a database containing a number of tablet computers available for purchase (Figure 1.1). Each tablet is represented by two attributes, its screen size and its battery longevity or autonomy. Intuitively, the most preferred tablets are those that have both of the two attributes as large as possible and,\(^4\)Throughout the thesis the terms “objects”, “tuples” and “points” are used interchangeably when referring to a dataset’s contents
1.1. Preference Queries

since the two attributes generally exhibit a trade-off between them, the overall problem can be characterized as a specific case of multi-criteria optimization in which the goal is to select the “best” tablet computers out of the entire dataset.

The majority of the queries and data structures presented in the following sections are based on the concept of “dominance” that is defined as follows.

**Definition 1 (Dominance).** Let \( S \) be a dataset of points. A point \( p \in S \) dominates another point \( q \in S \) \((p \prec q)\), iff \( p \) is as good as \( q \) in all dimensions and it is strictly better than \( q \) in at least one of the dimensions. A point \( p \) is a maxima point if it is not dominated by any point in \( S \).

Without loss of generality, in the above definition “better” corresponds to “larger”. Therefore, a point \( p \) dominates \( q \) \((p \prec q)\) when \( \forall i \in [1, d], \ p[i] \geq q[i] \) and \( \exists j : p[j] > q[j] \), where \( d \) is the total number of dimensions and \( p[i] \) is the value of \( p \) in the \( i \)-th dimension. The concept of dominance permits a form of comparison between two objects in a dataset by identifying which of the two objects is “strictly better”.

By utilizing the dominance concept elaborate queries can be defined that aim to select the most interesting or useful objects of a dataset (i.e. preference queries). The skyline query reports all the points in a dataset that are not dominated by any other point in the dataset, i.e. all maxima points.

**Definition 2 (Skyline Query).** The result of a skyline query, \( SKY(S) \) over a dataset \( S \) is composed of all points that are not dominated by any other point (i.e. all maxima points).

---

5Symmetrically, the definition also stands for when “better” corresponds to “smaller” in which a case a point \( p \) is a minima point if it is not dominated by any point in \( S \).
maxima points). Formally:

$$\text{SKY}(S) = \{ p \in S : \not\exists q \text{ s.t. } q < p \}$$

Each point found through a skyline query possesses the characteristic of not having an other point in the dataset with larger attribute values. Skyline queries are a simple and intuitive way of extracting interesting objects from a dataset, since they do not require a user-provided ranking function and are invariant of dimension scaling. Their drawback is that the query output may contain the entire dataset (or a single object) under specific data distributions and dimensionality thus resulting in the query output size not being controlled in general. As an example, the skyline points in Figure 1.1 are $p_4$ and $p_5$ since they are not dominated by any other point in the dataset.

On the other hand, top-k queries output the first $k$ objects with the largest score according to a user-defined ranking function.

**Definition 3 (Top-K Query).** The result of a top-k query over a dataset $S$, with respect to a specific scoring function, is composed of the $k$ points with the maximum scores, where $k$ is a user-provided parameter.

Contrary to skyline queries, the parameter $k$ permits top-k queries to have controlled output sizes. However, the scoring function employed is provided by the user and, depending on the application domain, may not always be straightforwardly defined in each case. The output of a top-2 query on the dataset of Figure 1.1 based on the scoring function $f(p) = p.x + p.y$ (sum of coordinates) is comprised of the points $p_4$ and $p_5$.

The top-k dominating query aims to combine the merits of the two other queries and was proposed in [96] and later studied in detail in [118, 119].

**Definition 4 (Dominance Score).** The dominance score $s(p)$ of a point $p$ is the number of points dominated by $p$. Formally:

$$s(p) = |\{ q \in S | p < q \}|$$ (1.1)

In essence, the top-k dominating query outputs the $k$ points with the largest score, where the score of each point corresponds to the amount of points it dominates. The top-k dominating query controls its output size through the parameter $k$ and avoids the use of a user-defined ranking function through the dominance score. Similarly to skyline queries, the result is also not sensitive to dimension scaling. In the running example of Figure 1.1 a top-2 dominating query would return the points $p_4$ and $p_2$ with dominance score 6 and 4 respectively.

Lastly, apart from efficient preference query evaluation, a research direction that has become prevalent in the presence of large and dynamic datasets is the practical maintenance of the “most interesting” objects when a dataset
1.2 Historical Information Management

An important challenge that arises with the dynamicity in modern networks (and their respective graphs) is the appropriate handling of their history so that features and properties that characterize the whole (or part of the) timeline of the graph can be extracted, as opposed to solely its latest state. Under this more encompassing approach, it is possible to answer elaborate queries such as “how has the diameter in a group of friends evolved between 2012 and today” in social networks or “how has the closeness centrality of author X progressed over time” in citation networks. In the former case, a decreasing diameter could signify that the members of the community are becoming more connected between them as time progresses while in the latter case an increasing closeness centrality could translate to the author becoming more “influential” as more works are published in the field.

A key aspect in effectively tackling the overall problem is the efficient storage of the evolving graph. The states of the evolving graph at different time instances are called snapshots and the graph is viewed as an evolving sequence of such snapshots.

**Definition 5 (Evolving Graph Sequence).** An evolving graph sequence \( \mathcal{G} \) is defined to be a collection of snapshots \( \mathcal{G} = \langle G_1, G_2, G_3, \ldots \rangle \). A graph snapshot \( G_i \in \mathcal{G} \) where \( G_i = (V_i, E_i) \), corresponds to the graph \( G \) at time instance \( i \) and is characterized by a set of vertices \( V_i \) and a set of incoming and outgoing edges \( E_i \).

The rate at which snapshots are obtained depends on the underlying network that the graph represents and is largely application-specific. Figure 1.2 depicts an evolving graph sequence \( \mathcal{G} \) that is composed of three snapshots \( G_1, G_2 \) and \( G_3 \) with each snapshot corresponding to the state of the graph \( G \) at time instances 1, 2 and 3 respectively. To obtain a particular snapshot from another snapshot in the sequence a set of operations has to be performed (e.g. \( G_2 \) can be obtained by adding an edge between \( a \) and \( c \) in \( G_1 \) and removing the edge between \( c \) and \( d \)).

Historical queries performed on evolving graph sequences may involve one or more of the sequence’s snapshots. The sets of operations (i.e., vertex/edge insertion, deletions and updates) between two consecutive snapshots are called deltas. A simple approach to solving the problem would be...
to explicitly store all snapshots separately. This strategy would be oblivious to the evolution of the graph in time which implies a prohibiting space overhead that also affects the time cost of the supported operations. A system that aims to efficiently store all the snapshots of such a graph should employ techniques that mitigate the presence of unaltered data between different snapshots (i.e. take advantage of the commonalities between snapshots and refrain from storing duplicate data across snapshots).

There have been two main approaches with regard to a system’s design [66], the time-centric approach and the entity-centric approach. In the former case the system is indexed according to the time instances (i.e. changes are organized by the time instance they occur in), while in the latter case the system is indexed according to the entities, their relationships and their respective history throughout the snapshots (i.e. changes are organized based on the vertex or edge they refer to).

Finally, another viewpoint concerning a system’s design is based on the type of queries that the system should be able to evaluate. Local queries are based on a particular vertex or a limited selection of vertices (e.g. the 2-hop neighborhood of a vertex) while global queries consider the majority or the entirety of a graph’s vertices (e.g. global clustering coefficient). Furthermore, both local and global queries should be able to be executed on either a single snapshot or on a range of snapshots. In the first case, a query aims to evaluate a measure at a particular time instance (e.g. shortest path length between two vertices at a particular time instance), while in the latter case a query’s objective is to extract information regarding a measure’s evolution through snapshots (e.g. average shortest path length between two vertices in the ten first snapshots). In general, systems built following the time-centric approach are more suited towards evaluating global queries, whereas local queries are expected to be handled more efficiently by systems following the entity-centric approach.
1.3 Motivation and Contributions

The primary focus of academic research in the field of algorithms and data structures design has been the development of optimized techniques that solve specific problems under efficient total running time and space cost. Historically, the main motivation for space-efficient data structures has been the restricted accessibility of large memory units due to their high prices. Nowadays, with the widely increased availability of economical and fast memory units, space-efficient solutions are designed to tackle issues that arise with the rapid data increase present in modern applications.

Typically, space-efficient data structures provide solutions that are able to solve larger problem instances in main memory. Additionally to this fact, they can handle larger base cases for classic divide and conquer algorithms that operate in external memory. These properties are important in the context of larger datasets that originate from sensor networks and contain temporal and spatial data. Another motivating factor for space-efficient solutions is the emergence of portable, handheld computing devices. These devices are usually smaller in scale and possess limited memory thereby making the space-efficient approach necessary. Finally, as a result of their low storage offprint, space-efficient solutions naturally exhibit greater locality of reference.

This thesis describes solutions to problems related to preference query evaluation as well as effective historical information management. An important common characteristic of all the algorithms and data structures that will be described in detail in the rest of the thesis is that they are asymptotically space-optimal, i.e. the space required by each one is linear with respect to the total dataset size.

With regard to preference queries, the thesis describes four data structures that dynamically maintain the top-$k$ dominating objects in a dataset under updates. More specifically, assuming a dataset of two dimensional points and a user-provided parameter $k$ the algorithms maintain the top-$k$ dominating points under two update settings. In the semi-dynamic setting only insertions of new points in the dataset are permitted, while in the fully-dynamic setting both insertions and deletions of points are supported. For each of the two update settings two variations of the algorithms are described which display a tradeoff between the update and query times.

Afterwards, the focus of the thesis shifts to skyline queries and the ML-Tree data structure. The ML-tree offers a method for finding skyline points in a two dimensional point dataset under point insertions and deletions. The points are drawn from specific distributions while the query region forms a 3-sided rectangle (which, in general, is more restrictive than the entirety of the dataset). For each returned point in a query, the ML-Tree can guarantee an expected loglogarithmic time cost with high probability (w.h.p.).

The second part of the thesis, which is concerned with historical information management, focuses on the analytical design and development of
an asymptotically space-optimal prototype system termed HiNode. HiNode follows an entity-centric design approach and moves away from the concept of using deltas to reconstruct specific snapshots. Essentially, it stores the history of a vertex as time intervals within the vertex itself and organizes it in such a form so that various operations (such as interval stabbing queries) are supported. The overall entity-centric and “local” approach to structuring the historical information naturally permits an efficient execution of local queries since only the parts of a sub-graph that are relevant to each query are reconstructed. These facts, along with the competent performance of HiNode in global queries that involve a range of snapshots, are experimentally demonstrated against the $G^*$ parallel graph processing system.

The prototype version of HiNode (termed $HiNode-G^*$) was built on top of the $G^*$ system by substituting its indexing module with the one proposed by HiNode. While the experimental results of the initial implementation showcased the efficiency of the entity-centric approach adopted by HiNode, this design choice incurred drawbacks regarding the efficiency and scalability of the prototype. To overcome the limitations of the $G^*$ storage engine, a second version of HiNode was implemented (termed $HiNode-NoSQL$) that moves away from the $G^*$ engine and adopts a NoSQL database approach as its underlying storage mechanism. Since the adoption of a NoSQL approach opens up different entity-centric modelling and storage approaches, two models were investigated and experimentally evaluated. The models’ performance was shown to differ by several times under certain queries and conditions demonstrating the fact that there doesn’t exist a single “best” model for all practical use cases. Finally, non-NoSQL solutions as well as simple baseline approaches were shown to be slower by up to an order of magnitude.

In summary, the thesis offers the following contributions:

- Four algorithms for the dynamic maintenance of top-$k$ dominating objects.
- Dynamic 3-sided skyline queries with an expected loglogarithmic time cost (w.h.p.) for each returned point.
- An asymptotically space-optimal vertex-centric system called HiNode for efficient graph historical information management.
- Two models for a NoSQL-based implementation of the proposed HiNode indexing module that exhibit better performance than non-NoSQL solutions.
1.4 Thesis Structure

Following the introduction, Chapter 2 performs a literature review on preference queries and historical information management. Chapter 3 describes the efficient maintenance of top-k dominating queries under updates with performance guarantees, while Chapter 4 focuses on the evaluation of dynamic planar 3-sided skyline queries for specific data distributions in expected time. Chapter 5 presents the initial implementation of the entity-centric system HiNode while Chapter 6 details its NoSQL extension and implementation. Finally, Chapter 7 concludes the thesis and presents directions for future work.
Chapter 2

Background and Literature Review

The process of adroitly selecting the most interesting objects or, generally, the ones that exhibit many desirable properties in a dataset has attracted considerable academic research interest over the past decades. Contrarily, the field of historical information management has become increasingly prominent over the past few years with the advent of massive and rapidly evolving networks such as those discussed in the introductory section. This chapter focuses on reviewing previous work conducted on the topics of both preference query evaluation as well as historical information management.

2.1 Preference Queries

Preference queries [3, 68] enable the acquisition of “relevant” objects in a dataset according to some desirable (and often contradictory) properties that each object must possess. Since the desirability of an object largely depends on the application at hand and can potentially be defined according to a user’s parameters, there have been a variety of preference query alternatives proposed over the years such as top-k queries [57], k-nearest neighbor queries [97], skyline queries [60], top-k dominating queries [96] and others. Since the contributions of the thesis mainly revolve around skyline and top-k dominating queries, this section details previous worked conducted on both of the two query types.

2.1.1 Skyline Queries

Skyline queries retrieve all objects in a dataset that are not dominated by other objects, i.e. each object for which there does not exist another object with better values in all dimensions. The origins of the skyline problem in the database community can be found in other fields of study, most notably those of Statistics (Admissible points [9]), Game Theory and Operations Research (Pareto optimality [90, 98]), and Computational Geometry (Maximal vectors [75]). In the following paragraphs $n$, $d$ and $k$ represent the dataset size, dataset dimensions and query output size respectively. All algorithms and data structures require linear space unless otherwise noted.
Kung et al. [75] described a skyline computation algorithm for 2-dimensional datasets that requires $O(n \log n)$ time. Additionally, they offered a solution for the general case of $d \geq 3$ that requires $O(n \log^{d-2} n)$ time while Bentley [11] proposed a divide and conquer algorithm that achieves the same time bounds of Kung et al. [75]. These algorithms are asymptotically optimal for $d = 2$ and $d = 3$ [75]. An output-sensitive solution of the problem was detailed by Kirkpatrick and Seidel [69] that achieves $O(n \log^{d-2} k)$ time for $d \geq 3$ and $O(n \log k)$ for $d = 2$. In the RAM model of computation, Gabow et al. [42] proposed a solution for $d \geq 4$ that requires $O(n \log^{d-3} n \log \log n)$ time with Chan et al. [27] later reducing the time cost to $O(n \log^{d-3} n)$. Additionally, Matousek [82] devised an algorithm that uses a matrix multiplication approach and achieves $O(n^{2.688})$ time for $d = n$.

The first definition of the skyline problem with respect to database systems was proposed by Börzsönyi et al. [17] through the use of the skyline operator. The authors argued that the above algorithms were not as practical and efficient for large, real-world datasets and they suggested two algorithms for the computation of a dataset’s skyline points: BNL and D&C. The Block-Nested-Loop (BNL) algorithm computes the cartesian product of a dataset’s tuples (i.e. compares each dataset tuple with every other tuple) and reports all tuples that are not dominated by any other tuple. In their Divide & Conquer (D&C) approach the original dataset is recursively subdivided in partitions until each partition can be processed in the main memory by itself. Then, the “local” skyline points are computed for each partition and the answers are merged to obtain the overall skyline set. The algorithms Sort-First-Skyline (SFS) [31] and its improvement Linear-Elimination-Sort-for-Skyline (LESS) [47] build on the BNL algorithm and include an initial point sorting step according to a function that is monotonically increasing or decreasing on all dimensions (e.g. the sum of a point’s coordinates on all dimensions). Each dataset point is assigned a score according to the function and the ensuing scanning of the points in a sorted order guarantees that at any moment a point cannot be dominated by subsequent points, resulting in a potential reduction of pairwise comparisons.

While the above algorithms achieve efficient running times for several practical use-cases and datasets, their asymptotical performance in the worst case is characterised by quadratic time. Another approach to computing skylines is through the use of preconstructed indices on the original datasets such as the algorithms Nearest Neighbor (NN) [74] and Branch-and-Bound-Skyline (BBS) [96] that employ $R^*$-trees [10, 49] to refrain from performing unnecessary pairwise comparisons. While such approaches generally perform competently in practice, they too exhibit quadratic time in the worst case.

In the dynamic version of the problem, the goal is to maintain the set of
skyline points under insertions and deletions of points in the dataset. Overmars and van Leeuwen [93] first described a data structure in the Pointer Machine computation model that was able to maintain the skyline points of a 2-dimensional (planar) dataset under insertions and deletions in $O(\log^2 n)$ update time and $O(k)$ query time. In their solution, the points on the skyline set are stored in concatenable queues that support split and merge operations. Whenever an insertion or a deletion of a point occurred, the data structure would perform $O(\log n)$ split and merge operations between the affected queues with $O(\log n)$ cost each, in order to recompute the skyline points.

The solution of Overmars and van Leeuwen served as the basis for several works that improved on the update times. Frederickson and Rodger improved the insertion cost to $O(\log n)$ [39] (see also [59] for a similar result by Janardan). D’Amore et al. [32] focused on boundary updates (i.e. the point that is inserted or deleted has the largest or smallest x coordinate out of all the points in the dataset) and for this special case achieved insertion and deletion time of $O(\log n)$. Kapoor [61] attained $O(\log n)$ update time for arbitrary points with a query cost of $(\text{chng} \times \log n + k)$ where chng is the number of points updated since the last skyline reporting query. Following that, Brodal and Tsakalidis [22] achieved $O(\log n)$ worst-case update time and $O(k)$ query time. Additionally, in the word-RAM computation model [40] they also proposed a data structure that requires $O(\frac{\log n}{\log \log n})$ update time and $O(k)$ query time [22].

Finally, it’s worth noting that over the past years, skyline queries have also been studied in an assortment of different contexts and variations such as skyline for datasets following specific point distributions [13, 12, 33], k-dominant skylines [26], k-representative skylines [80], skyline computation in metric spaces [30], uncertain skylines [1], $\epsilon$-skylines [114] and others.

### 2.1.2 Top-k Dominating Queries

A basic method for retrieving the top-k dominating points of a dataset would consist of, firstly, computing the dominance score of each point and then using a linear time selection algorithm [15] to find the point $v$ with the k-th largest score. To find all the top-k dominating points a final scan on the dataset is performed and all points with a greater score than the score of $v$ are reported.

The simplest approach for computing the domination score for all points would be to compare each point $p$ with every other point $q$ in the dataset and increment $p$’s score if it dominates $q$. This results in $O(n^2)$ time cost and $O(n)$ space cost. An approach with lower time complexity would be to use a 2-dimensional range counting data structure (e.g., [29, 54]). For each point $p = (x_p, y_p)$ in a dataset $S$, the points lying in the query rectangle
Chapter 2. Background and Literature Review

Q = \([x_p, \infty) \times [y_p, \infty)\) can be counted in \(O(\log n)\) time and \(O(n)\) space using the 2-dimensional range counting data structure by Chazelle [29]. The number of points found in Q is equal to \(p\)'s dominance score. In order to compute the dominance score of each point, the process is repeated for all the points in \(S\) in \(O(n \log n)\) total time. Lastly, an algorithm by Chan and Pătraşcu [28] is able to compute the dominance score for all points in \(O(n^{1/2} \log n)\) time in the word-RAM model [40] of computation. Insertions and deletions can be trivially supported in the above methods in \(O(n)\) time since one has to update the dominance scores of all points in the worst-case. In the following, more elaborate methods for answering a top-\(k\) dominating query are described.

Papadias et al. [96], first proposed the \(d\)-dimensional top-\(k\) dominating query along with a solution based on the iterative computation of a dataset’s skyline points. More specifically, they observed that the top-1 dominating point of a dataset is contained in the dataset’s skyline points. This stems from the observation that for every point \(p\) not in the skyline, there exists a point \(p'\) in the skyline that dominates it and, as a result, \(p'\) has a larger score than \(p\). Thus, in their approach, they compute the set of skyline \(M\) (using the BBS algorithm [96]) and compute the dominance score of all the points in \(M\). The point \(q\) with the highest score is the top-1 dominating point and is thereby reported. Finally, \(q\) is removed from the dataset and the procedure is repeated until \(k\) points have been reported. However, this approach does not avoid the quadratic trap, since the score computation of skyline points as well as the update of dominance scores after the removal of the point with the highest score, may lead to \(O(n^2)\) dominance checks, whereas the space remains linear\(^1\).

Yiu and Mamoulis [118, 119] recommended using aggregate R-trees (aR-trees) to efficiently compute \(d\)-dimensional top-\(k\) dominating queries. They provided various algorithms based on aR-trees that proved experimentally to be quite fast. They also make an analytic study making the assumption that the data points are uniformly and independently distributed in a domain space. The authors do not make any statement for the worst-case time complexity of the query but it is certainly \(\Omega(n)\).

Both methods [96, 119] focus on the top-\(k\) dominating query, where \(k\) is arbitrary. Update operations can be applied in both cases with a linear time cost. However, the top-\(k\) dominating query has to be re-evaluated in both cases. Finally, both prove the efficiency of their approach experimentally (extensive experiments can be found in [119]).

As a closing remark, it’s worth noting that top-\(k\) dominating queries have also been studied in the context of data streams [72], uncertain databases [79, 121], spatial objects [115] and vertically decomposed data [110].

\(^1\)The BBS algorithm is based on the use of R-trees which require linear space.
2.2 Historical Information Management

An evolving graph sequence can be formed by periodically collecting the state of a graph at various time instances. As a result, additional insight on the nature of each originating network can be obtained by incorporating temporal aspects in the traditional graph processing methods. Current centralized and distributed graph processing systems such as Pregel [81], Neo4j [85], Trinity [105], Giraph [46] and others focus on processing single and very large graphs without supporting temporal extensions to the typical graph processing queries. As a result, these systems are not inherently suitable for performing analysis on evolving graph sequences.

Most of the research conducted towards maintaining evolving graph sequences aims to exploit the commonalities that exist between a graph in different time instances in order to improve space or time efficiency. The work performed in the area is in an inceptive stage and thus solutions for both centralized and parallel or distributed approaches are presented. Among the centralized methods is the FVF framework by Ren et al. [100] that groups the sequence graphs into clusters and operates on them. Another method was proposed by Koloniari et al. [71] and it is based on maintaining a log of operations (defined as deltas) that occur in the graph between various time instances and employing it to reconstruct the graph at a particular time instance. Caro et al. [18, 25] proposed space-efficient methods that utilize compact and self-indexed data structures to reduce the total space cost. Finally, methods have been proposed [5, 56, 116, 104] that index the sequence in a manner that permits the efficient evaluation of certain queries.

In the parallel and distributed setting there have been three main methods proposed: The DeltaGraph system [65] (along with its extension HGS [67]) is based on the principle of deltas and aims to efficiently store and retrieve the graph at specific time instances. The G* system [76, 107, 77] is a parallel graph database that focuses on taking advantage of the commonalities present between a graph in different time instances to store the sequence in an efficient manner. Finally, the GreyCat system [51] adopts a multiple-world approach to the evolution of graph sequences that permits large scale what-if analysis.

2.2.1 Centralized Methods

At the start, the FVF framework by Ren et al. [100] will be described followed by the works of Koloniari et al. [71] and Caro et al. [25]. Following that, the section will conclude by discussing indexing methods for evolving graph sequences that tackle certain historical queries.
Chapter 2. Background and Literature Review

The FVF Framework

The first centralized method reviewed is the FVF (FIND - VERIFY - FIX) framework proposed by Ren et al. [100]. The authors describe a method that consists of two phases, a preprocessing phase and a query-processing phase, and additionally propose storage models for the evolving graph sequences that support the aforementioned framework.

In the preprocessing phase the initial snapshots of the sequence are gathered into smaller clusters of similar snapshots. This is performed by defining a graph similarity measure and by incrementally adding snapshots in a cluster (starting from the first snapshot in the sequence) until a graph similarity threshold has been surpassed. At that point, a new empty cluster is created and the above procedure is repeated until all the snapshots have been examined. For each cluster, two representative graphs $G_\cap$ and $G_\cup$ are extracted which are the largest common subgraph and the smallest common supergraph of all snapshots in the cluster respectively. For example, assuming that $G_1$ and $G_2$ from Figure 1.2 are grouped in the same cluster their respective $G_\cap$ and $G_\cup$ graphs correspond to the graphs in Figure 2.1.

In the query-processing phase the authors use the clusters and their representative graphs to answer shortest path and closeness centrality queries. At first they evaluate the solution to a query for the representative graphs of the cluster (“FIND” step) on the basis that the solution will readily apply to a number of the snapshots in the cluster. In the “VERIFY” step, the evaluated solution is tested with each snapshot in the cluster in conjunction with a set of intuitive lemmas. For each snapshot that the evaluated solution does not apply, the framework attempts to “FIX” the solution so that it also applies to the aforementioned snapshot.

The authors also propose three storage models that can be used along with the FVF framework. The models make use of the similarities exhibited between successive snapshots and between representative graphs of successive clusters to reduce the total space cost of the evolving graph sequence. Finally, they assess their work through extensive experiments on both real and synthetic datasets.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure2.1.png}
\caption{$G_\cup$ and $G_\cap$ for the evolving graph sequence in Figure 1.2.}
\end{figure}
Using Graph Deltas for Historical Queries

The authors in [71] advocate the use of graph deltas to support historical queries on evolving graph sequences. They begin by stating the operations that are supported on each snapshot, namely, \textit{addNode}(u_i), \textit{remNode}(u_i), \textit{addEdge}(u_i, u_j), \textit{remEdge}(u_i, u_j) which correspond to the addition or removal of a vertex \( u_i \) and the addition or removal of an edge between two vertices \( u_i \) and \( u_j \) respectively. Graph deltas are defined to be sets of such operations that when applied on a particular snapshot they yield another snapshot of the sequence. For example in Figure 1.2, \( G_3 \) can be obtained by applying \( \{ \textit{addNode}(e), \textit{addEdge}(a, e) \} \) to \( G_2 \).

Furthermore, they define complete deltas to be sets of operations that when applied on the first snapshot of the sequence they are able to yield any of the sequence’s snapshots\(^2\). Additionally, inverted deltas are defined to be sets of operations that when applied on a snapshot \( G_t \) they yield a snapshot \( G_{t'} \) where \( t' < t \), that is, \( G_{t'} \) occurs “earlier” in the sequence than \( G_t \).

Having defined the different types of deltas, the authors discuss snapshot materialization techniques and policies. More specifically, while any of the sequence’s snapshots may be reconstructed if a complete and invertible delta and another one of the sequence’s snapshots are maintained, it may be to the method’s benefit to also maintain interposed snapshots to speed up snapshot materialization.

The next body of the work proposes three plans for efficient query processing. Perhaps the most universal of the proposed plans is a two-phase query plan that first materializes a particular snapshot according to the techniques discussed and then executes the query on the materialized snapshot. Finally, the authors investigate potential optimizations, delta indexing approaches and present some preliminary results of their solutions.

Compact Sequence Representations

Until this point the previous work discussed was focused on reducing the total time cost of queries on evolving graph sequences. In the following works by Caro et al. [25] the authors address the problem of reducing the space cost when handling evolving graph sequences. Their proposed methods are heavily based on compact and self-indexed data structures that coupled with certain compression techniques (such as ETDC [19] and the PForDelta technique [120, 122]) achieve overall high space efficiency with an acceptable trade-off on the total time cost of the queries.

The authors use the concept of contacts as described by Nicosia et al. [87] to define temporal graphs\(^3\). A contact is defined to be a 4-tuple \((u, v, t_s, t_e)\)
that signifies the existence of an edge between vertices $u$ and $v$ during the time period $[t_s, t_e]$. The collection of all contacts is equivalent to the temporal graph itself, while, a particular snapshot $G_t$ corresponds to the set of contacts $(u, v, t_s, t_e)$ such that $t \in [t_s, t_e]$.

Next, operations that can be performed upon temporal graphs are presented. Those include:

- neighbor queries (i.e. report all neighbors of a vertex $u$),
- reverse neighbor queries (i.e. report all vertices that have a vertex $u$ as neighbor),
- active edge queries (i.e. does there exist an edge between two vertices $u$ and $v$ at time instance $t$?),
- retrieving a snapshot of the graph at time instance $t$,
- edge state change queries (i.e. report all edges that have had their state changed at time instance $t$, that is all contacts that $t_s = t$ or $t_e = t$)

After a brief overview of the compression techniques and compact data structures they use in their work, the authors focus on the four temporal graph representations they propose along with their implementations that take advantage of the compression techniques. The first representation, that is named EdgeLog, is an index that maintains for every vertex $v$ in the temporal graph a list with the neighbors of $v$. Each neighbor of $v$ is also equipped with a list containing all the time intervals that the particular edge exists in the sequence. The EdgeLog structure for a sequence composed by the graphs $G_1$ and $G_2$ of the example in Figure 1.2, is depicted in Figure 2.2.

The second representation, called EveLog, follows a similar approach to the first. More specifically, EveLog is composed of a list with all the vertices
that appear in the temporal graph. For each vertex \( v \), there exists a list with all the “events” related to \( v \) (i.e. edge state change along with the vertex at the other end of the edge). The third representation is titled Compact Adjacency Sequence (CAS) and is based on the use of the Wavelet tree, while the fourth representation (CET) is based on the Interleaved Wavelet tree which is a data structure proposed in the same work as an additional asset to handling temporal graphs.

The work is concluded with extensive experimental evaluation over synthetic and real datasets through which the authors reach an interesting conclusion that there isn’t a single best data structure for all the queries performed on temporal graphs. As a last remark, it should be noted that the above work focuses on the offline version of the problem, yet it also mentions alterations and modifications that need to be done in order for the solutions to apply to the online version. In the offline version the entire sequence \( \mathcal{G} \) is known beforehand and update operations are not supported in any snapshot (i.e. \( \mathcal{G} = \langle G_1, G_2, G_3 \rangle \) only consists of \( G_1, G_2 \) and \( G_3 \) and no new snapshots are created). In the online version, \( \mathcal{G} \) is constantly evolving and is not characterized by a “final” snapshot (i.e. \( \mathcal{G} = \langle G_1, G_2, G_3, \ldots \rangle \) may eventually end up with more than three snapshots).

Constructing Indices for Specific Queries

The work presented so far mostly focuses on efficiently storing, maintaining and retrieving the snapshots of an evolving graph sequences. There have been methods proposed in literature that instead aim to index the evolving graph sequence in a manner that permits the effective evaluation of specific queries. A number of examples are presented in the paragraphs that follow.

Akiba et al. [5] describe dynamic indexing schemes that permit them to answer distance queries on either the last snapshot (current) or in any “older” snapshot in the sequence. Furthermore, they support the historical distance change-point query that reports all the time instances in the sequence where the distance between two vertices \( u \) and \( v \) changes. It is worth noting that in their work, they handle graphs that only support vertex additions and edge additions.

An other method that concentrates on answering shortest path queries was proposed by Huo et al. [56]. The authors make use of a Temporally Evolving Graph structure to store all the updates that occur in the sequence and proceed to use variations of Dijkstra’s algorithm [34] to compute shortest paths. Furthermore, they speed up their solutions by making use of preprocessing indexes, namely, Contraction Hierarchies [45].

Yang et al. [116] detail an algorithm that discovers most frequently changing components in an evolving graph sequence. They begin by defining measures of change between vertices and the general problem of extracting the most frequently changing component and proceed to present their solutions.
Finally, Semertzidis et al. [104] tackle the problem of answering historical reachability queries. Their proposed index structure is called TimeReach and it is built in a manner that takes advantage of the strongly connected components that are present in a graph.

2.2.2 Parallel and Distributed Methods

In this section the attention turns to systems that were proposed for parallel and distributed environments. The systems that will be analyzed are the DeltaGraph system by Khurana et al. [65], the G* graph database by Labouseur et al. [76, 107, 77] and the GreyCat system by Hartmann et al. [51].

The DeltaGraph and Historical Graph Store Systems

Khurana et al. [65] designed and implemented a distributed system called DeltaGraph aiming to efficiently store and retrieve snapshots from an evolving graph sequence. DeltaGraph supports time point (singlepoint) queries, time interval snapshot queries and multiple time point (multipoint) queries. Furthermore, along with the graph structure a query is also able to return the attributes of vertices and edges (e.g. name, weight etc.) The system is composed of two main components: the DeltaGraph index structure and the GraphPool in-memory data structure.

The DeltaGraph index is described as a rooted hierarchical graph structure that resembles a tree with adjacent leaves connected to each other in a bidirectional manner. The leaves of the structure correspond to snapshots of the sequence while the inner nodes correspond to graphs that can be obtained by applying a differential function (e.g. Intersection) to its children. The edges between the nodes store sets of deltas that are used to obtain a child node from its parent and they are horizontally partitioned between workers. It should be noted that the only data stored are the sets of deltas and not the graphs themselves although the authors advocate the materialization of specific snapshots in DeltaGraph so as to speed up query time.

To answer a singlepoint query for a time instance $t$, the system locates through a binary search among the leaves the two adjacent leaves that “encompass” the query point $t$. Afterwards, it finds the minimum-weight path from the root to either of the two leaves, where the weight of an edge is set to be equal to the size of its respective delta. For multipoint queries, the system follows the same procedure with the difference being that instead of finding a path with minimum weight the system has to find the lowest-weight Steiner tree between the root and the multiple time instances.

The other component of the system is the GraphPool data structure which maintains in-memory a combination of materialized snapshots. More specifically, GraphPool maintains the current graph, historical snapshots and materialized graphs in a single combined graph. To determine which graphs
contain a certain component or attribute the system makes uses of a mapping table. Finally, GraphPool is responsible for keeping the current graph index updated and cleaning up historical snapshots that are no longer needed.

The authors built on and extended the core principles of DeltaGraph by implementing the Historical Graph Store (HGS) system [67]. HGS is composed of two primary modules: The first module is the Temporal Graph Index (TGI) that enhances the indexing capability of DeltaGraph with the addition of a meta index that maintains historical changes on a per-node basis and permits the querying of smaller subgraphs compared to entire snapshots. Additionally, HGS contains the Temporal Graph Analysis Framework (TAF) that enables the execution of various complex data analysis tasks.

The G* Graph Database

The G* graph database by Labouseur et al. [76, 107] focuses on taking advantage of the commonalities that exist between snapshots in a sequence so that they are stored in an efficient manner.

In the G* system, each server is assigned a set of vertices along with all the outgoing edges of each vertex in the set. This achieves significant data locality since obtaining all of a vertex’s edges can be accomplished without the need to contact any of the other servers. Furthermore, since the snapshots in a sequence exhibit similarities between them, G* avoids storing redundant information by only storing each version of a vertex once and, in that way, data that isn’t modified between snapshots isn’t needlessly stored again.

Additionally, each server maintains an index named Compact Graph Index (CGI) that stores a single \((vertexID,disk\_location)\) pair for each vertex version that exists in a combination of the sequence’s snapshots. For example, the CGI of a server maintaining vertex \(c\) of Figure 1.2 would contain two pairs related to \(c\): A pair for version \(c_1\) in \(\{G_1\}\) and another pair for version \(c_2\) in \(\{G_2,G_3\}\). It should be noted that the CGI has a low space overhead and can be mostly or fully kept in memory. As a last remark, the authors have proposed splitting the CGI in a specific manner when a large number of graph combinations has been formed in its contents.

Similarly to the storage module of G*, the CGI can also be used with regard to query processing to ensure that each version of vertex or edge is only processed once per query evaluation. Furthermore, the G* system supplies three types of primitives that can be used to construct graph query operators: summaries, combiners and bulk synchronous parallel (BSP) operators. Finally, snapshot replication and distribution techniques are discussed in [77].

The GreyCat system

The last system that will be reviewed is the GreyCat system by Hartmann et al. [51]. The authors introduce the Many-Worlds Graphs (MWG) data model
that is inspired by the many-world interpretation of Everett [37] and exemplified by Schrödinger’s cat [103]. In their model every operation results in a divergence point that produces a new, separate world with the overall objective being the efficient support of large scale what-if analytics.

The authors first discuss the motivation behind their work through a real-world case study before moving on to the description of the model’s key concepts. Essentially, a triplet consisting of a node n, a timepoint t and a world w, uniquely identifies the state of a node (i.e. the attributes and relationships of the node for the values in that particular triplet). States are arranged into chunks which correspond to the storage units of GreyCat whereas a series of ordered chunks for a particular node and world form a timeline. Following these definitions, the authors gradually define the MWG model by starting with a base graph definition and then incorporating temporal elements as well as the many-worlds semantics.

Next, the authors focus on implementation details regarding the GreyCat system itself. More specifically, they delve deeper into the process of mapping nodes to state chunks as well as their storage. Additionally, they study indexing, concurrency, consistency and distribution issues and provide examples of their system APIs for manipulating the MWG model. Finally, the work is concluded with experimental evaluation on a practical case study as well as compared to state of the art systems.

An overview of the works presented above can be seen on Table 2.1.
Chapter 3

Dynamic Top-k Dominating Queries

This work is the first attempt to provide efficient algorithms for top-k dominating query processing in the semi-dynamic and the fully-dynamic cases, which are the most interesting and challenging. In contrast to previously proposed techniques (Section 2.1.2), this work is concerned with proposing algorithms with non-trivial performance guarantees.

Let $S$ be a dataset of $n$ 2-dimensional points. One may think that perhaps a direct application of a “divide and conquer” algorithmic technique could provide an efficient solution at least for the static top-k dominating query, where given the dataset $S$ the goal is to retrieve the $k$ points with the highest domination scores. The problem with this approach is that the top-k dominating query is a non-decomposable query, because the score of each point depends on the coordinates of all the other points in $S$. A query $q$ in $S$ is decomposable [14] if its output can be computed accurately by executing $q$ in a partition of $S$. The non-decomposability of top-k dominating queries prohibits the use of standard divide-and-conquer techniques and thus increases the problem difficulty significantly.

This chapter concentrates on 2-dimensional data for two reasons. First, there is no previous work with asymptotic guarantees and as a result, this work provides a deeper understanding of the complexity of the problem. The second, more practical, reason is that many applications are inherently 2-dimensional. This is because, one often faces the situation of having to strike a balance between a pair of naturally contradicting factors (e.g., price vs quality, space vs query time). Finally, the algorithms are based on a novel restricted dynamization of layers of minima [16]. This is of independent interest in case only the first $k$ layers of minima are of interest.

Since static datasets are being handled rarely by modern applications, the chapter considers the problem in the semi-dynamic case (insertions only), where logarithmic complexities are attained. In the fully-dynamic case, polynomial complexities are attained for update operations (insertions and deletions). In many applications, insertions occur much more frequently than deletions. As a practical example, consider an application that retrieves the
### Table 3.1: Chapter 3 contributions. SD stands for Semi-Dynamic, where only insertions are allowed, whereas FD stands for Fully-Dynamic where both insertions and deletions are supported. Worst-case times are marked with “w.c.” while amortized times are marked with “am.” All algorithms use linear space.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Preprocessing Cost (worst-case)</th>
<th>Query Cost (worst-case)</th>
<th>Update Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD/(k)-list</td>
<td>(O(n \log n))</td>
<td>(O(k))</td>
<td>(O(\log^2 n + k^2 \log n)) w.c.</td>
</tr>
<tr>
<td>SD/1-list</td>
<td>(O(n \log n))</td>
<td>(O(k \log n))</td>
<td>(O(\log^2 n + k \log n)) w.c.</td>
</tr>
<tr>
<td>FD/(k)-list</td>
<td>(O(n \log n))</td>
<td>(O(k))</td>
<td>(O((k + \sqrt{n})k \log n)) am.</td>
</tr>
<tr>
<td>FD/1-list</td>
<td>(O(n \log n))</td>
<td>(O(k \log n))</td>
<td>(O((k + \sqrt{n}) \log n)) am.</td>
</tr>
</tbody>
</table>

Top-\(k\) dominating tweets (i.e., Twitter messages) according to some attributes that are user-selected (e.g. number of retweets, number of the author’s followers etc.). In this application, the semi-dynamic algorithms would suffice since tweets very rarely are deleted [6]. Other possible examples of datasets where insertions take place significantly more often than deletions include the measurements collected by a scientific instrument or the full-year sales log of a retail company. In conclusion, applications where deletions occur orders of magnitude less frequently than insertions can benefit from the use of the semi-dynamic algorithms and the associated data structures.

For each of the semi-dynamic and fully dynamic settings two solutions (\(k\)-list and 1-list) are provided that exhibit a trade-off between update and query time. All the algorithms use linear space and work well under the realistic assumption that \(k\) is a fixed user-defined parameter which is small compared to the size \(n\) of the dataset (i.e., \(k \ll n\)). Table 3.1 provides a detailed overview of the results.

The rest of the chapter is organized as follows. Section 3.1 presents some necessary concepts related to the discussion that follows. The contribution for the semi-dynamic case is detailed in Section 3.2, whereas the study of the fully-dynamic case is offered in Section 3.3. In addition to the results for the RAM computation model, in Section 3.4 an adaptation to the word-RAM model is provided, that obtains better asymptotic bounds.

### 3.1 Preliminaries

In this section, we discuss the basic concepts that are used throughout the rest of this work. First of all, we define a point \(p_i\) to be a triplet of the form \(p_i = (x_i, y_i, s_i)\) where \(x_i, y_i\) are its \(x\) and \(y\) coordinates and \(s_i = s(p_i)\) is its dominance score (see Definition 1).

In the following two sections, we describe the concept of *layers of minima* and we cite a previous result in the form of a lemma, that will be used in the query phase of some of the proposed solutions.
3.1 Preliminaries

3.1.1 Two-dimensional Layers of Minima

The algorithms presented in the remaining sections are based on the concept of layers of minima. In order to compute the layers of minima of a dataset $S$ we perform a skyline query on $S$, remove the answer set of points from $S$ and repeat the process until no points remain in $S$. The set that results from the $i$-th skyline query forms the $i$-th layer of minima. By collecting all the layers, we form the layers of minima of $S$. A concise definition of the layers of minima follows:

**Definition 6.** Let $M_1$ be the set of all minima points in $S$. The first layer of minima of $S$ is equal to the set $M_1$ and the second layer of minima $M_2$ of $S$ is equal to the set of all minima points in $S − M_1$. The $j$-th layer of minima of $S$ is accordingly defined to be equal to the set of all minima points in $S − (\bigcup_{i=1}^{j-1} M_i)$. The sequence $< M_1, M_2, \ldots, M_\lambda >$ where $S = \bigcup_{i=1}^{\lambda} M_i$ is the layers of minima of $S$.

Figure 3.1 depicts a layer of minima on the plane. Any point located in the shadowed region is dominated by at least one point in the layer of minima.

Blunck and Vahrenhold [16] proposed in-place algorithms that use $O(1)$ extra space and compute the layers of minima of a dataset of 2-dimensional points in $O(n \log n)$ time.

3.1.2 Reporting Lemma

Finally, we use the following lemma from [38]:

**Figure 3.1:** The first layer of minima (skyline).
Lemma 1. Let $A_1, \ldots, A_m$ be arrays of values from a totally ordered set such that each array is sorted. Given an integer $L \leq \sum_{i=1}^m |A_i|$, there is a comparison-based algorithm that finds in $O(m)$ time a value $\tau$ that is greater than at least $L$ but at most $O(L)$ values in $A_1 \cup \ldots \cup A_m$.

Lemma 1 can be adjusted to report a value $\tau$ that is smaller than at least $L$ but at most $O(L)$ values in $A_1 \cup \ldots \cup A_m$. This lemma forms the basis in the process of efficiently finding the $k$-th point with the highest score out of a collection of ordered lists and is used in Sections 3.2 and 3.3.

3.2 The Semi-Dynamic Case

In this section, we propose a solution to the semi-dynamic top-$k$ dominating query problem and describe in detail the data structures and algorithms we use to achieve it. Let $S$ be a set of $n$ 2-dimensional points. Recall that the semi-dynamic top-$k$ dominating query aims at reporting the $k$ points in $S$ with the highest dominance score where $k$ is a user-defined parameter that is fixed at the time when the data structure is constructed. Furthermore, $S$ is subject to insertions of new points. This poses an additional challenge since after inserting a new point, it is possible that the dominance score of many (or even all) the points in $S$ must be updated. Individually updating the score of each such point would be computationally prohibitive so we follow a different approach and only update lazily the score of groups of points that are candidates for being in the final answer.

We first note that when a point $p$ dominates another point $q$, $p$’s score is strictly greater than the score of $q$:

$$\forall p, q \in S, p \preceq q \Rightarrow s_p > s_q$$ (3.1)

Organizing $S$ into layers of minima offers an intuitive way of using the above property to eliminate points that are not possible to belong in the final answer. As an example, consider a top-1 dominating query in $S$. The point with the highest dominance score is found in the first layer of minima of $S$ since all the points in the second and subsequent layers are dominated by at least one other point. Similarly, in a top-2 dominating query, the first point is found in the first layer and the second point is found in either the first or the second layer. In general, the following lemma holds for the top-$k$ dominating points:

Lemma 2. The top-$k$ dominating points of $S$ are located in the first $k$ layers of minima of $S$.

Proof. If $S$ has only $k$ or less layers of minima, the lemma obviously holds. Otherwise, assume that a point $p$ belongs to the $i$-th layer of minima, where $i \geq k + 1$. There are at least $i - 1$ points dominating $p$ and due to Equation
3.1 all of them have a larger score than \( p \). As a result, \( p \) is not included in the top-\( k \) dominating points of \( S \).

A direct consequence of Lemma 2 is that, when inserting a new point \( p \), we only need to update the scores of some points in the first \( k \) layers of minima. However, some of the layers may have many points and thus individually updating the score of these points would result in a high update cost. To avoid this, after inserting a new point \( p \) in \( S \), we find only the first and last point that dominate \( p \) in each layer. This pair of points denotes an interval that marks all the points in each layer whose score must be updated. Consequently, by examining only \( O(1) \) points in each layer the total update cost is reduced.

Lastly, an issue brought up by the use of layers of minima is that the insertion of a new point \( p \) may create cascading changes to the structure of the layers. In particular, by inserting \( p \) into \( S \), \( p \) must also be inserted in one of the layers of \( S \). Let \( L_i \) be that layer. The insertion of \( p \) in \( L_i \) may cause some of its points to be discarded as a result of them being dominated by \( p \). This group of points must be inserted into the next layer \( L_{i+1} \) possibly discarding some of the points in \( L_{i+1} \) in the process. Due to Lemma 2 and the fact that only insertions are allowed, this chain of operations only has to be performed up until the \( k \)-th layer.

To achieve efficient insertion, we model each layer as an \((a, b)\)-tree. In the following, we provide a detailed overview of the data structure and the operations it supports and then we describe the update and query algorithms for the semi-dynamic top-\( k \) dominating query.

### 3.2.1 The Augmented \((a, b)\)-Tree

We model each layer of minima using an augmented leaf-oriented \((a, b)\)-tree. Let \( L \) be a layer of minima containing \( m \) points, i.e., \( p_1, p_2, \ldots, p_m \) where \( p_i = (x_i, y_i, s_i), 1 \leq i \leq m \). Since the points in \( L \) are totally ordered on each dimension\(^1\), we can use a single \((a, b)\)-tree to search among the points in both dimensions. To achieve that, each inner node stores representative keys for both dimensions, instead of storing keys for only one of them. We use \( h_v \) to denote the height of a node \( v \) in the tree.

For each node \( v \) of the tree with \( h_v \geq \log_b k \) we maintain a field \( \text{add}(v) \). The field’s contents denote a score that has to be added to the score of all the points in \( v \)'s subtree. Finally, each node \( v \) with \( h_v \geq \log_b k \) is augmented with a \( k \)-sized list \( \text{top}(v) \) which stores the \( k \) points with the highest score in \( v \)'s subtree. An example of an augmented \((a, b)\)-tree node with \( \text{height} > \log_b k \) for \( k = 2 \) can be seen in Figure 3.2.

\(^1\)For two points \( p_a = (x_a, y_a, s_a) \) and \( p_b = (x_b, y_b, s_b) \) in \( L \) if \( x_a > x_b \) then \( y_b > y_a \)
Figure 3.2: An \((a,b)\)-tree node \((k = 2)\). The coordinates in node \(v\) are designated by the respective representative points \(p_i, 1 \leq i \leq b - 1\), where \(p_i\) is the leftmost point of the \(i + 1\)-th child of \(v\).

For the remainder of this work we assume that \(b = O(1)\) since we present main memory algorithms. The following lemma provides the tree’s total space cost.

**Lemma 3.** The total space cost of an augmented \((a,b)\)-tree storing \(m\) points is \(O(m)\).

**Proof.** All the nodes with height lower than \(\log_b k\) only store \(O(1)\) additional information so their total space cost is \(O(m)\). There are \(O(m/k)\) nodes with height higher than or equal to \(\log_b k\) each augmented with a \(k\)-sized list. The total space cost of this part of the data structure is \(O(m/k) \times O(k) = O(m)\). As a result, the total space cost of the entire data structure is \(O(m)\).

The following lemma provides the time complexity for the construction of an augmented \((a,b)\)-tree.

**Lemma 4.** The construction of an augmented \((a,b)\)-tree over \(m\) points that are sorted according to their dimensions can be carried out in \(O(m \log k)\) time, where \(k\) is a user-defined parameter.

**Proof.** In order to construct the leaf-oriented augmented \((a,b)\)-tree we follow a bottom-up approach and assume that the input points are sorted according to their dimensions. The augmented \((a,b)\)-tree is constructed in a similar way to a typical \((a,b)\)-tree with an additional issue. At first, the nodes of the augmented \((a,b)\)-tree are constructed by scanning the input points, creating the leaves and then recursively creating the inner nodes from bottom to top. Each node is only visited once so the procedure up to this point requires \(O(m)\) time.

The last step is to compute the \(\text{top}\) lists for all nodes with \(h_v \geq \log_b k\). For each node \(v\) with \(h_v > \log_b k\), the \(\text{top}(v)\) list must be computed from the \(\text{top}\) lists of \(v\)’s children. By simultaneously traversing the \(O(b) = O(1)\) \(\text{top}\) lists of \(v\)’s children we can compute \(\text{top}(v)\) in \(O(k)\) time. There are \(O(m/k)\) nodes with \(h_v > \log_b k\) and since this process is repeated for every node, the time required is \(O(m/k) \times O(k) = O(m)\).
Finally, we compute the top lists for each node \( v \) with \( h_v = \log_b k \). Since \( v \)'s children are not augmented with top lists we follow a different approach. We sort all the points found in \( v \)'s subtree\(^2\) in \( O(k \log k) \) time and store them in top \( (v) \). There are \( O(m/k) \) nodes with \( h_v = \log_b k \) and thus this step requires \( O(m \log k) \) total time.

Thus, the total time required for constructing the \((a, b)\)-tree is \( O(m \log k) \).

\[ \square \]

## Operations

In this section, we formally describe all the operations supported by the augmented \((a, b)\)-tree. More specifically, the augmented \((a, b)\)-tree supports searching for a point, inserting a new point, or deleting an existing one. Furthermore, splits and concatenations between two different \((a, b)\)-trees are also supported.

The search operation search \((T, p_z)\) locates the augmented \((a, b)\)-tree \( T \) a specific point \( p_z \) and can be performed with respect to either dimension of \( p_z \) by using the appropriate set of keys. Let \( v \) be a node of \( T \), \( x_1, x_2, \ldots, x_{b-1} \) be the \( x \)-representative keys of \( v \)'s children and \( y_1, y_2, \ldots, y_{b-1} \) be the \( y \)-representative keys of \( v \)'s children. In order to search for a point \( p_z = (x_z, y_z, s_z) \) in \( T \), we begin at the root and search down until we reach a leaf. If the search is performed on the \( x \) dimension, we select the \( i \)-th child of \( v \) such that \( x_{i-1} < x_z \leq x_i \). Otherwise, if the search is performed on the \( y \) dimension, we select the \( i \)-th child of \( v \) such that \( y_{i-1} > y_z \geq y_i \). Since \( T \) is height-balanced, a search operation requires \( O(b \log m) = O(\log m) \) time.

The rest of the operations are based on node splits and merges. For reasons of clarity, we first describe how node splits and node merges are handled on the augmented \((a, b)\)-tree in relation to typical \((a, b)\)-trees.

The node split operation node_split \((v, v_1, v_2)\) is executed similarly to the split operation of typical \((a, b)\)-trees with a few modifications. More specifically, before dividing a node \( v \) into two nodes \( v_1 \) and \( v_2 \) we check the contents of add \((v)\). If add \((v)\) stores a value different than 0, we add the contents of add \((v)\) to the add variable of \( v \)'s children and set add \((v)\) to 0. Afterwards, \( v \) is divided into \( v_1 \) and \( v_2 \) and the keys for the \( x \) and \( y \) dimensions of \( v \) are “divided” between \( v_1 \) and \( v_2 \) in \( O(b) = O(1) \) time. After sharing the keys, top \((v_1)\) for \( v_1 \) and top \((v_2)\) for \( v_2 \) must be recomputed. If \( h_v > \log_b k \) we can compute top \((v_1)\) and top \((v_2)\) in \( O(k) \) time by simultaneously traversing the \( O(b) = O(1) \) top lists of \( v_1 \)'s and \( v_2 \)'s children respectively. As a result, the split operation requires \( O(k) \) time in this case.

If \( h_v = \log_b k \) then the children of \( v_1 \) and \( v_2 \) are not augmented with top lists and thus the computation of top \((v_1)\) and top \((v_2)\) cannot be performed using the above procedure. Each of \( v_1 \) and \( v_2 \) have up to \( k \) points in their

\[ \text{\textsuperscript{2}There are up to } k \text{ points in } v \text{'s subtree since } h_v = \log_b k \]
subtree since their height is equal to \( \log_b k \). To compute the top lists for \( v_1 \) and \( v_2 \) we sort all the points in \( v_1 \)'s and \( v_2 \)'s subtree in \( O(k \log k) \) time and store the result in \( \text{top}(v_1) \) and \( \text{top}(v_2) \) respectively. This results in the split operation requiring \( O(k \log k) \) time in this case. Finally, if \( h_v < \log_b k \) then the split operation requires \( O(b) = O(1) \) time since \( v \) is not augmented with \( \text{top}(v) \).

For the merge operation \( \text{node}\_\text{merge}(v_1, v_2, v) \) we follow a similar procedure to the merge operation of standard \((a, b)\)-trees. More specifically, before merging two nodes \( v_1 \) and \( v_2 \) into \( v \) we check the contents of \( \text{add}(v_1) \) and \( \text{add}(v_2) \). If \( \text{add}(v_1) \) stores a value different than 0 we add the contents of \( \text{add}(v_1) \) to \( v_1 \)'s children and set \( \text{add}(v_1) \) to 0. We follow the same procedure for \( \text{add}(v_2) \). Then, \( v_1 \) and \( v_2 \) are merged into \( v \) and the keys for the \( x \) and \( y \) dimensions of \( v \) are derived from the keys of \( v_1 \) and \( v_2 \) in \( O(b) = O(1) \) time. After merging the keys, \( \text{top}(v) \) must be recomputed. To achieve this, if \( h_v > \log_b k \) we simultaneously traverse \( \text{top}(v_1) \) and \( \text{top}(v_2) \) and store the \( k \) points with the highest score in \( \text{top}(v) \) in \( O(k) \) time. If \( h_v = \log_b k \) we follow a similar approach to that of a node split and first compute \( \text{top}(v_1) \) and \( \text{top}(v_2) \) by sorting all the points in \( v_1 \)'s and \( v_2 \)'s subtree respectively. Then by simultaneously traversing \( \text{top}(v_1) \) and \( \text{top}(v_2) \) we store the \( k \) points with the highest score in \( \text{top}(v) \). As in the split operation, if \( h_v < \log_b k \) the merge operation requires \( O(b) = O(1) \) time due to the fact that \( v \) is not augmented with \( \text{top}(v) \).

Operation \( \text{insert}(T, p) \), inserts a point \( p \) in \( T \). The point is inserted as a leaf in \( T \) and the tree is rebalanced using node splits. Since there are \( O(\log m) \) node splits that cost \( O(k) \) time and \( O(1) \) splits that cost \( O(k \log k) \) time, the time cost to insert a point is \( O(k \log m) \).

Operation \( \text{delete}(T, p) \), removes a point \( p \) from \( T \). The leaf corresponding to the point is removed and the resulting tree is rebalanced. There are \( O(\log m) \) merges that cost \( O(k) \) time, \( O(1) \) merges that cost \( O(k \log k) \) time and a possible terminating split and as a result the time cost to delete a point is \( O(k \log m) \).

Using the node split and node merge operations as building blocks, we can define two additional operations on the augmented \((a, b)\)-trees: Tree Concatenation and Tree Split. For both the operations, we use the definition and algorithms provided in [83].

Operation \( \text{concat}(T_1, T_2, T_3) \), concatenates two augmented \((a, b)\)-trees \( T_1 \) and \( T_2 \) into a third augmented \((a, b)\)-tree \( T_3 \). This operation assumes that \( \max\{T_1\} \leq \min\{T_2\} \) where \( \max\{T_i\} \) is the largest \( x \) coordinate of all the points in \( T_i \) and \( \min\{T_i\} \) is smallest \( x \) coordinate of all the points in \( T_i \) (a similarly defined order is implied for the \( y \) dimension as well). In a tree concatenation one merge operation and up to \( O(\log \max(|T_1|, |T_2|)) \) split operations are performed. Since there are \( O(1) \) merge and split operations that cost \( O(k \log k) \) time and the rest of the merge and split operations cost \( O(k) \) time,
a tree concatenation operation requires \(O(k \log \max(|T_1|, |T_2|))\) time. Before initiating this operation, all \(add\) variables in the affected path are flashed to their children (get a zero value).

Lastly, operation \(\text{split}(T_1, val, T_2, T_3)\) splits an augmented \((a, b)\)-tree \(T_1\) into two augmented \((a, b)\)-trees \(T_2\) and \(T_3\) at element \(val\) with respect to the one of the two dimensions, so that \(T_2 \leftarrow \{z \in T_1 ; z \leq val\}\) and \(T_3 \leftarrow \{z \in T_1 ; z > val\}\). In a tree split operation the starting \((a, b)\)-tree is first split into two forests of trees. Then, the roots of the trees in each forest are merged with each other recursively. Splitting the tree into two forests requires \(O(\log |T_1|)\) time and since there are \(O(1)\) merge operations that cost \(O(k \log k)\) time and \(O(\log |T_1|)\) merges for both forests, each requiring \(O(k)\) time, a tree split operation requires \(O(k \log |T_1|)\) time. Similarly to \(\text{concat}\), before initiating this operation, all \(add\) variables in the affected path are flashed to their children. The following theorem summarizes the discussion on the \((a, b)\)-tree.

**Theorem 1.** Given a parameter \(k\) and \(m\) 2-dimensional points \(p_i = (x_i, y_i, s_i)\) where \(1 \leq i \leq m\), we can construct in \(O(m \log k)\) time an augmented \((a, b)\)-tree \(T_1\) that uses \(O(m)\) space. The construction time assumes that the points are sorted according to their dimensions. The tree \(T_1\) supports the following operations:

- \(\text{search}(T_1, p)\) in \(O(\log m)\) time,
- \(\text{insert}(T, p)\) and \(\text{delete}(T, p)\) in \(O(k \log m)\) time,
- \(\text{split}(T_1, val, T_2, T_3)\) in \(O(k \log m)\) time and
- \(\text{concat}(T_1, T_4, T_5)\) in \(O(k \log \max(|T_1|, |T_4|))\) time (we note that \(T_4\) is an augmented \((a, b)\)-tree such that the condition \(\max\{T_1\} \leq \min\{T_4\}\) holds).

### 3.2.2 Insertion

Let \(p = (x_p, y_p, s_p) \in \mathbb{R}\) be a point to be inserted into \(S\). Furthermore, let \(L_1, \ldots, L_k\) be the first \(k\) layers of minima of \(S\). Before inserting \(p\) we compute its dominance score using the dynamic range counting data structure proposed in [29]\(^3\). We also insert \(p\) in the dynamic range counting data structure in order to support score computation for future insertions. The data structure supports queries and updates in \(O(\log^2 n)\) time and \(O(n)\) space.

Next, we find if \(p\) must be inserted in one of \(L_1, \ldots, L_k\) by searching each of the \(k\) respective \((a, b)\)-trees for \(p\). Starting from \(L_1\) and iterating towards \(L_k\), we search each tree for \(p\) both in the \(x\) and in the \(y\) dimension and retrieve the predecessor of \(p\) in the \(x\) dimension and the predecessor of \(p\) in the \(y\) dimension. If neither of those two points dominate \(p\), we insert \(p\) in the tree’s respective layer and stop the iteration. Otherwise, the iteration may

\(^3\)The data structure is built only once as a preprocessing step before the first insertion.
end without any layer satisfying the above condition. In that case, $p$ does not become a member of any of the $k$ first layers.

If we do not insert $p$ in any of the $k$ first layers then we only have to update the scores of some points in each of $L_1, \ldots, L_k$. Otherwise, assume that $p$ is inserted into $L_i$ where $1 \leq i \leq k$. Then we have to update scores of points in $L_1, \ldots, L_{i-1}$ and alter the structure of $L_i, \ldots, L_k$ (Figure 3.3). We first describe how to handle score updating on a layer and afterwards how to alter a layer’s structure using tree splits and tree concatenations.

To update the score of the points in layer $L$, we perform this procedure. We search the augmented $(a, b)$-tree of $L$ for $x_p$ and $y_p$. All points whose score must be updated lie to the left of $x_p$ and to the right of $y_p$. Since updating the score of each point would be time consuming, we only find the two boundary points that define the above interval and mark the subtrees between them. For example, in Figure 3.3 the insertion of $p$ causes a score increment for the points $p_{t-2}^i$ to $p_{t-2}^b$ in layer $L_{i-2}$ and the points $p_{t-1}^i$ to $p_{b-1}^b$ in layer $L_{i-1}$.

We start from height $= \log_b k + 1$ of the two search paths and move up towards the root, adding $+1$ to $add(v)$ if $v$ is a node hanging immediately to the left of the search path for $x_p$ or immediately to the right of the search path for $y_p$. Using this method we denote that the score of all the points in $v$’s subtree must be incremented by one, without actually visiting the points themselves. Adding $+1$ to $add(v)$ does not change $top(v)$ since we increment the score of all the points in $v$’s subtree and thus their relative order according to score remains unchanged. For each node $v'$ on the search paths or hanging on the search paths with $h_{v'} = \log_b k$, instead of incrementing $add(v')$, we exhaustively check the points in the subtree of $v'$ and individually update.
3.2. The Semi-Dynamic Case

\[ L_{i+1} \]

\[ L_i \]

\[ p = (x_p, y_p) \]

Insertion of \( p \)

\[ T \]

\[ 1) \text{Split}(T, y_p, T_1, T_2) \]

\[ p = (x_p, y_p) \]

\[ T_1 \]

\[ T_2 \]

\[ T_c \]

\[ T_{\text{new}} \]

\[ 2) \text{Split}(T_2, x_p, T_3, T_c) \]

\[ p \]

\[ T_3 \]

\[ T_1 \]

\[ T_c \]

\[ T_{\text{new}} \]

\[ 3) \text{insert}(T_3, p) \]

\[ 4) \text{Concat}(T_3, T_1, T_{\text{new}}) \]

\[ T_c \text{ is now the input to } L_{i+1} \]

**FIGURE 3.4:** Example of layer restructuring.

their score based on if they are dominating \( p \). Finally, we sort the points in the subtree of \( v' \) based on the updated scores and store the result in \( \text{top}(v') \).

For each node with \( \text{height} < \log_b k \) no action is necessary since all the points in its subtree can be found in the \( \text{top} \) list of its ancestor with \( \text{height} = \log_b k \). Thus, at the end, we have indirectly marked all the points between \( y_p \) and \( x_p \) for score increment.

Finally, we update the \( \text{top} \) lists of the nodes in the search path as a result of modifying the \( \text{add} \) fields of their children. Starting from \( \text{height} = \log_b k + 1 \) and moving towards the root, we recursively compute the \( \text{top} \) list of each node \( v \) by simultaneously merging the \( \text{top} \) lists of its children. While merging the lists, we also add the contents of each node’s \( \text{add} \) field to the score of the node’s \( \text{top} \) list points so as to take into account the score changes caused by the insertion of \( p \). At the end, the \( \text{top} \) list found in the root of the \((a, b)\)-tree will have the correct top-\( k \) points for that layer of minima.

Indirectly marking all the points between \( y_p \) and \( x_p \) for score increment requires \( O(\log n + k \log k) \) total time while merging the top-\( k \) lists of a node’s children requires \( O(bk) = O(k) \) time and as a result the total cost for all the
nodes in the search paths of the tree is $O(bk \log n) = O(k \log n)$ time. Thus, the total time required to update scores in a layer is $O(k \log n)$.

In the second case, the point $p$ may have to be inserted in layer of minima $L_i$. Since inserting or deleting points from the layer of minima one-by-one would be time consuming, we insert the point and remove the now-dominated points by executing a series of tree splits and tree concatenations. First, we find the interval of points dominated by $p$ as previously by querying the layer of minima tree $T$ for $x_p$ and $y_p$. Then we perform the following sequence of operations in order:

1. $\text{split}(T, y_p, T_1, T_2)$,
2. $\text{split}(T_2, x_p, T_3, T_c)$,
3. $\text{insert}(T_3, p)$ and
4. $\text{concat}(T_3, T_1, T_{\text{new}})$.

Recall that all add variables of nodes on the affected paths for operations $\text{split}$ and $\text{concat}$ are flashed to their children, that is in all such paths the add variables have zero value. The effects of this sequence of operations on the layers of minima can be seen in Figure 3.4. The layer of minima tree $T_{\text{new}}$ for $L_i$ now correctly has $p$ inserted and every point previously in $L_i$ that is now dominated by $p$ (i.e., $T_c$) has been discarded.

If $T_c$ is empty, the iteration stops. Otherwise, $T_c$ is propagated to the next layer of minima $L_{i+1}$ where we repeat the above procedure with $T_c$ as the input. Since $T_c$ may have more than one points, instead of inserting them one-by-one we perform a tree concatenation at step (3) instead of an insertion. Finally, the insertion spot of $T_c$ in $L_{i+1}$ can be found by querying the tree of $L_{i+1}$ for $p' = (x'_p, y'_p)$ where $x'_p$ is the $x$ coordinate of the leftmost point in $T_c$ and $y'_p$ is the $y$ coordinate of the rightmost point in $T_c$.

In Figure 3.3, the insertion of $p$ causes the points $p^i_l$ to $p^i_b$ to be discarded from layer $L_i$. These points are propagated to layer $L_{i+1}$ and will, in turn, discard the points $p^{i+1}_l$ to $p^{i+1}_b$ from $L_{i+1}$. This procedure is repeated until the $k$-th layer of minima.

In each layer of minima we perform a series of $O(1)$ splits and concatenations. Since each tree split or concatenation requires $O(\log n)$ time, the total time required to alter a layer’s structure given a point or an $(a, b)$-tree as an input is $O(k \log n)$.

As described in the beginning of the section, after an insertion a layer must either update the score of some of its points or alter its structure. Since either case requires $O(k \log n)$ time, the time cost of manipulating the $k$ first layers after an insertion is $O(k^2 \log n)$. Adding the cost of computing the score of the inserted point and inserting the point in the dynamic range counting data structure, the total insertion cost is $O(\log^2 n + k^2 \log n)$ time.
3.2.3 Query

To find the top-\(k\) dominating points of \(S\), we apply Lemma 1 for \(L = k\) on all the top-\(k\) lists found in the root of each \((a, b)\)-tree of each of the \(k\) first layers of minima. Let \(I\) be the list returned by Lemma 1. By selecting the \((|I| - k + 1)\)-th order statistic\(^4\) of \(I\) we obtain the dominance score \(\tau\) of the \(k\)-th top dominating point. Finally, we traverse all the top-\(k\) lists we previously collected and report all points with score larger than \(\tau\). Since the lists are sorted according to their score, we can stop traversing a list when a point with score lower than \(\tau\) has been found. Applying Lemma 1 requires \(O(k)\) time while finding the \((|I| - k + 1)\)-th order statistic of \(I\) requires \(O(I) = O(k)\) time. Finally, traversing all lists requires \(O(k)\) time in total. By combining all of the above, we achieve \(O(k)\) query time.

3.2.4 Reducing the Update Cost

We can reduce the algorithm’s update cost by shrinking the size of the top list in each node of each \((a, b)\)-tree. In particular, we store a top list in each node of the \((a, b)\)-tree but instead of storing \(k\) points in each list we only store 1. This removes the cost of computing top lists during each node split or merge since each top list can be computed using \(O(b) = O(1)\) comparisons. As a result, node splits and merges cost \(O(1)\) time and updating the score of points in a layer or altering its structure costs \(O(\log n)\) time. This brings the total insertion cost down to \(O(\log^2 n + k \log n)\) time.

This change also implies that at the time of a query, each \((a, b)\)-tree’s root only stores 1 element with the highest score in the layer and as a result we can no longer directly apply Lemma 1. To overcome this we build a Strict Fibonacci Heap [21] by inserting each point with the highest score from each layer. Strict Fibonacci Heaps support insertions in \(O(1)\) worst-case time and deletions of the maximum key in \(O(\log n)\) worst-case time. By querying the heap we are able to find (and delete) the top-1 dominating point. After deleting a point \(p\) (belonging in a layer \(L\)) from the heap, we have to replace it by the point of \(L\) with the next highest score. This point can be found by querying \(L\)’s tree for \(p\). Due to the definition of top lists, the point with the next highest score in \(L\) is guaranteed to be amongst the \(O(b)\) top lists of each node in the search path of \(p\). We insert all \(O(b \log n) = O(\log n)\) such points in the heap and repeat the process until \(k\) points have been deleted from the heap. In order to not have any duplicate points in the heap, we also employ a marking process.

The deletion of a point from the heap requires \(O(\log n)\) time, while adding \(O(\log n)\) points also requires \(O(\log n)\) time. Since there aren’t any duplicate points in the heap and the process is repeated \(k\) times, the query phase of the algorithm requires \(O(k \log n)\) time.

\(^4\)The \(i\)-th order statistic of a set of \(n\) elements is the \(i\)-th smallest element in the set
Finally, we analyze the preprocessing cost in both semi-dynamic algorithms. In the case of the \(k\)-list augmented \((a, b)\)-tree the construction time is equal to \(O(n \log n)\). To achieve this, we build Chazelle’s static range counting data structure \([29]\) in \(O(n \log n)\) time and count the score of each point using the method we described in Section 2.1.2. We also build Chazelle’s dynamic range counting data structure \([29]\) which is required by the insertion algorithm in \(O(n \log n)\) time. The layers of minima can be computed in \(O(n \log n)\) time \([16]\). A subsequent scan of the output provides us with the points of each layer of minima in sorted order and as a result we can build the \((a, b)\)-trees in \(O(n \log k)\) time (Lemma 4). Therefore, the construction time is equal to \(O(n \log n)\). In the case of the 1-list augmented \((a, b)\)-tree the only difference is the construction cost of the \((a, b)\)-trees which is reduced to \(O(n)\) since each node in each \((a, b)\)-tree is augmented with a top list of size 1. The discussion of this section can be summarized in the following theorem:

**Theorem 2.** Given a set of \(n\) 2-dimensional points, we can build a data structure that supports insertions of new points in \(O(\log^2 n + k^2 \log n)\) worst-case time and top-\(k\) dominating queries in \(O(k)\) worst-case time. Alternatively, we can build a data structure that supports insertions of new points in \(O(\log^2 n + k \log n)\) worst-case time and top-\(k\) dominating queries in \(O(k \log n)\) worst-case time. Both data structures are built in \(O(n \log n)\) time and use \(O(n)\) space.

### 3.3 The Fully-Dynamic Case

The algorithms presented so far only support insertions due to the fact that all operations could be restricted in the first \(k\) layers of minima of a dataset \(S\). However, assume the deletion of a point \(p\) in layer \(L_k\). Then we would have to store and manipulate more than \(k\) layers since it is possible that some points from \(L_{k+1}\) might have to be inserted in \(L_k\) as a result of them not being dominated by any other point in \(L_k\) apart from \(p\). This brings a cascading of restructuring operations since some of the points in \(L_{k+2}\) might have to be inserted in \(L_{k+1}\). Thus, a deletion operation may reach the last layer of \(S\) in the worst-case. It should be noted that a deletion of a point may not always result in layer restructuring. Consider the example in Figure 3.5. Deleting \(p_d\) will cause the layers to be restructured since \(p_f\) is not dominated by any other point. However, deleting \(p_e\) will not cause any changes to the layers’ structure since both \(p_g\) and \(p_h\) are dominated by at least one other point in \(p_e\)’s layer.

A deletion of an existing point can be defined in a similar way to the insertion of a point with each layer requiring either score updating or restructuring. To perform score updating we follow the same steps as those discussed in Section 3.2.2 but instead of adding +1 to the add field of a node, we add −1. After deleting a point \(p = (x_p, y_p)\) from a layer \(L_i\), we query \(L_{i+1}\) to find
3.3. The Fully-Dynamic Case

all the points (if any) that must be inserted in $L_i$ due to the deletion of $p$. The query point in $L_{i+1}$ is $p' = (x'_p, y'_p)$ where $x'_p$ is the successor of $x_p$ in $L_i$ and $y'_p$ is the successor of $y_p$ in $L_i$.

The algorithms for the semi-dynamic setting can be extended to the fully dynamic setting by making use of the global rebuilding technique [94]. More specifically in an update operation, instead of manipulating only the first $k$ layers we perform score updates and layer restructuring operations in the first $k + \sqrt{n}$ layers. Since we stop restructuring operations on a predefined point, after the $i$-th deletion the $(k + \sqrt{n} - i + 1)$-th layer will have become invalid. As a result, after $\sqrt{n}$ deletions, only the first $k$ layers remain valid and at that point we rebuild the entire layers of minima data structure. We also recompute the score of each point and reconstruct the $(a, b)$-trees.

The following theorem analyzes the cost of the global rebuilding operation.

**Theorem 3.** The global rebuilding cost of the data structures is $O(n \log n)$ time and $O(n)$ space.

**Proof.** The result is derived from Theorem 2 and the discussion prior to it. Note that the dynamic range counting data structure is only built once (in the preprocessing phase) and it is not built again in any global rebuilding process applied to the data structures.

We perform the global rebuilding step once in every $\sqrt{n}$ updates. An update up to the $(k + \sqrt{n})$-th layer, requires $O(\log^2 n + (k + \sqrt{n})k \log n)$ time. We perform $\sqrt{n}$ such updates and then we globally rebuild the data structures in $O(n \log n)$ time so the amortized time for an update over $\sqrt{n}$ updates is $O(\sqrt{n} \log n + (k + \sqrt{n})k \log n) = O((k + \sqrt{n})k \log n)$.
Chapter 3. Dynamic Top-k Dominating Queries

The global rebuilding technique can also be applied on the method of Section 3.2.4 to obtain a data structure that handles insertions and deletions with reduced update cost. The results in this section are outlined in the following theorem:

**Theorem 4.** Given a set of $n$ 2-dimensional points, we can support updates in $O((k + \sqrt{n})k \log n)$ amortized time and top-$k$ dominating queries in $O(k)$ time. Alternatively, we are able to support insertions and deletions in $O((k + \sqrt{n}) \log n)$ amortized time and top-$k$ dominating queries in $O(k \log n)$ time. Both data structures are constructed in $O(n \log n)$ time and use $O(n)$ space.

### 3.4 Results for Word-RAM

In the previous results we have focused on the RAM model of computation. We can obtain slightly faster update algorithms for the semi-dynamic algorithm we presented by extending the results to the *word-RAM* model of computation. In the unit-cost *word-RAM model* [40], the memory is represented as an array of infinite cells (words) with each word storing $w$ bits. The input elements are considered to be integers from the universe $\mathbb{U}^2 = \{0, \ldots, 2^w - 1\}^2$ so that any word can be addressed by any other word (through the use of a pointer).

The model supports random access of words as well as comparisons, arithmetic, shift and bitwise operations between words in constant time. In this work, we make the assumption that $w = \Theta(\log n)$ where $n$ is the input’s dataset size. This fact permits an input point or an index to the data structure to fit in a single word. The space cost under the word-RAM model is defined with respect to the number of words occupied; while the query and update times with respect to the number of word accesses and comparisons or operations needed to answer a query or perform an update respectively. The word-RAM model is a realistic model of computation, with integers of bounded precision, that closely emulates the mechanics of many programming languages [28].

To obtain the results, we use the dynamic range counting data structure of He and Munro [53] which, for word size $w = \Omega(\log n)$, supports queries in $O((\log n / \log \log n)^2)$ worst-case time, insertions and deletions in $O((\log n / \log \log n)^2)$ amortized time and uses $O(n)$ space.

The construction cost of the proposed data structure is equal to the cost of constructing the dynamic range counting data structure, computing the score of each point, computing the layers of minima and constructing the $(a, b)$-trees. To build the dynamic range counting data structure we insert each point in the data structure for a total of $O(n(\log n / \log \log n)^2)$ amortized time. The score of all points can be computed using the data structure in $O(n(\log n / \log \log n)^2)$ total amortized time. The layers of minima can be built in $O(n \log n)$ time.
Lastly, we can build the \((a, b)\)-trees in \(O(n \log k)\) time using Lemma 4. Apart from the dynamic range counting data structure, the insertion and query algorithms remain the same. Combining the above observations we obtain the following result.

**Theorem 5.** Given a set of \(n\) 2-dimensional points in the word-RAM model with word size \(w = \Theta(\log n)\), we can build a data structure that supports insertions of new points in \(O\left(\frac{\log n}{\log \log n}\right)^2 + k^2 \log n\) amortized time and top-k dominating queries in \(O(k)\) worst-case time. Alternatively, we can build a data structure that supports insertions of new points in \(O\left(\frac{\log n}{\log \log n}\right)^2 + k \log n\) amortized time and top-k dominating queries in \(O(k \log n)\) worst-case time. Both data structures are constructed in \(O(n \left(\frac{\log n}{\log \log n}\right)^2)\) amortized time and use \(O(n)\) space.
Chapter 4

Dynamic 3-sided Skyline Queries

This chapter studies efficient algorithms with non-trivial performance guarantees for dynamic planar skyline processing. Let $P$ denote the set of points in the dataset. Also, let $p_i$ denote the value of the $i$-th coordinate of a point $p$. Recall that, the skyline of a set of points $P$ contains the points that are not dominated by any other point. In addition to the skyline requirements, there may exist additional constraints in the form of boundary restrictions on the dimension values themselves. For example, in the tablet computer database example of Figure 1.1 the user may only be interested in tablet computers with a screen size that is strictly larger than a specific size. Usually, these additional constraints form a rectangular area referred to as the region of interest. The answer to the query comprises the skyline of the points residing inside the region of interest.

An example range skyline query is given in Figure 4.1, where semantics of the form $(\max(X), \max(Y))$\textsuperscript{1} are being used. The skyline inside the region of interest is composed of the full black dot points and any hollowed points inside the query range are dominated by at least one of the region’s skyline points.

This work presents the ML (Modified Layered Range) tree-structure that provides a loglogarithmic expected time (per point) solution for finding planar skyline points in a 3-sided query rectangle in the RAM model under point insertions and deletions. This form of query resembles a 3-sided range reporting query with an additional skyline requirement and is handled by the ML-Tree for points drawn from specific distributions in $O(\log^2 N \log \log N)$ expected update time and $O(t \log \log N)$ query time with high probability. The proposed data structure is inspired from the Modified Priority Search Tree presented in [64] that supports 3-sided range reporting queries. However, the modifications performed to support skyline queries are non-trivial.

The best previous solution was presented in [22] and supports range skyline queries in $O(\frac{\log N}{\log \log N} + t)$ worst case time and updates in $O(\frac{\log N}{\log \log N})$ worst case time using linear space in the word-RAM model of computation with word size $w$, where the coordinates of the points are integers in the

\textsuperscript{1}Preferable points are characterized by large values in the $x$ dimension and large values in the $y$ dimension
range \( U = 0, \ldots, 2^w - 1 \) (see Section 2.1.1). Although their solution is optimal in the general case, ML-tree achieves better query time when the size of the reported skyline is small (approximately \( t < \frac{\log N}{\log^2 \log N} \)) and when the point coordinates follow specific class distributions.

The rest of the chapter is organized as follows. Fundamental concepts are presented in Section 4.1 while the ML-tree and its dynamic version are given in Sections 4.2 and 4.3, respectively.

### 4.1 Fundamental Concepts

For the remainder of this chapter we adhere to the RAM model of computation. We denote by \( N \) the number of elements that reside in the data structures and by \( t \) the size of the query. We begin by discussing the probability distributions that the algorithms and data structures are based on and then move on to describing the data structures that we use in order to achieve the desired complexities.

In the update operations we will consider that the points to be inserted are drawn by an unknown discrete distribution. Also, the asymptotic bounds are given with respect to the current size of the data structure. Finally, deletions of the elements of the data structures are assumed to be uniformly random. That is, every element present in the data structure is equally likely to be deleted [70].
4.1.1 Probability Distributions

In this section, we overview the probabilistic distributions that will be used in the remainder of the chapter. A probability distribution is $\mu$-random if the elements are drawn randomly with respect to a density function denoted by $\mu$. For the purpose of this work, we assume that $\mu$ is unknown.

Informally, a distribution defined over an interval $I$ is smooth if the probability density over any subinterval of $I$ does not exceed a specific bound, however small this subinterval is (i.e., the distribution does not contain sharp peaks).

Given two functions $f_1$ and $f_2$, then $\forall x \in U, \mu(x)$ is $(f_1, f_2)$-smooth if there exists a constant $\beta$, such that for all $c_1, c_2, c_3 \in U : c_1 < c_2 < c_3$, and for all naturals $\nu \leq n$, for a random key $x \in U$ it holds that:

$$\Pr \left[ c_2 - \left\lfloor \frac{c_3 - c_1}{f_1(\nu)} \right\rfloor \leq x \leq c_2 | c_1 \leq x \leq c_3 \right] = \sum_{c_2 - \left\lfloor \frac{c_3 - c_1}{f_1(\nu)} \right\rfloor}^{c_2} \mu_{[c_1, c_3]}(x) \leq \frac{\beta f_2(\nu)}{\nu}$$

where $\mu_{[c_1, c_3]}(x) = 0$ for $x < c_1$ or $x > c_3$, and $\mu_{[c_1, c_3]}(x) = \mu(x) / p$ for $x \in \{c_1, \ldots, c_3\}$ where $p = \sum_{c_1}^{c_3} \mu(x)$ and $p > 0$.

The above imply that no key can get a point mass, i.e. a value with nonzero probability. More accurately, if we initially consider the whole universe of keys with $|U| = n^\delta, c > 1$, and $\forall \nu \leq n$, we equally split it into $f_1(\nu) = \nu^\alpha, \alpha < 1$, many equal consecutive subsets of keys, then the above equation implies that each subset (containing $\geq \frac{|U|}{f_1(\nu)} = \frac{\nu^\delta}{\nu^\alpha} = \omega(1)$ consecutive keys) gets probability mass $\leq \frac{f_2(\nu)}{\nu^\delta}, \forall \nu \leq n$, which is $o(1)$ as $n \rightarrow \infty$, when $f_2(n) = n^\delta, \forall \delta \in (0, 1)$. Hence, as $n \rightarrow \infty$, each key in $U$ has $o(1)$ probability mass. Once more, we can describe the above equation by rephrasing the intuitive description of $(f_1, f_2)$-smooth distribution as:

“among a number (measured by $f_1(n) = n^\alpha, \alpha < 1$) of consecutive subsets, each containing consecutive keys from $U$, no subset containing consecutive keys from $U$ should be too dense (measured by $f_2(n) = n^\delta, \delta < 1$) compared to the others”.

The class of $(f_1, f_2)$-smooth distributions (for appropriate choices of $f_1$ and $f_2$) is a superset of both regular and uniform classes of distributions, as well as of several non-uniform classes [7, 113, 63]. Actually, any probability distribution is $(f_1, \Theta(N))$-smooth, for a suitable choice of $\beta$.

4.1.2 Auxiliary Data Structures

This section provides a concise overview of the data structures and algorithms that will be used as building blocks for the ML-tree in the following sections. More specifically, we will describe (half-)Range Minimum/Maximum Queries, $q^\alpha$-heaps and Interpolation Search Trees.

---

2In the sense that it is bounded below by a positive constant.
Half-Range Maximum Queries: The half-Range Maximum Query (h-RMQ) problem asks to preprocess an array \( A \) of size \( N \) such that, given an index range \([r, N]\) where \( 1 \leq r \leq N \), we are asked to report the position of the maximum element in this range on \( A \). Notice that we do not want to change the order of the elements in \( A \), in which case the problem would be trivial. This is a restricted version of the general RMQ problem, in which the range is \([r, r']\), where \( 1 \leq r \leq r' \leq N \). In [50] the RMQ problem is solved in \( O(1) \) time using \( O(N) \) space and \( O(N) \) preprocessing time. We could use this solution for the h-RMQ problem, but for this case the problem can be solved much simpler by maintaining an additional array \( A_{\text{max}} \) of maximum elements for each of the \( N \) positions in the initial array.

\( q^* \)-heaps: The \( q^* \)-heap [112] is a data structure having the following property: let \( M \) be the current number of elements in the \( q^* \)-heap and let \( N \) be an upper bound on the maximum number of elements ever stored in the \( q^* \)-heap. Then, update and query operations are carried out in \( O(1 + \log M \log \log N) \) worst-case time after an \( O(N) \) preprocessing overhead. The \( q^* \)-heap uses linear space and is constructed in linear time.

Interpolation Search Trees: In [62], a dynamic data structure was presented that supports insertion/deletion in \( O(1) \) time w.c. as well as predecessor/successor queries in \( O(\log \log N) \) expected time w.h.p., given that the keys are drawn from a \((N^\alpha, N^\beta)\)-smooth distribution, where \( 0 < \alpha, \beta < 1 \). It requires linear space.

### 4.2 The ML-Tree

In the following, we describe in detail the indexing scheme, which is termed as the Modified Layered Range Tree (ML-tree). The description of the ML-tree is considered for \((\max(X), \max(Y))\) skyline points. This static data structure for the problem servers only as a step towards the dynamic solution and does not provide better complexities in total for the static case when compared to what is currently known.

#### 4.2.1 The Static Non-Linear-Space ML-tree

The static Non-Linear ML-tree is stored as an array \( A \) in memory, yet it can be visualized as a complete binary tree. The static data structure is an augmented binary search tree \( T \) on the set of points \( S \) that resembles a range tree. \( T \) stores all points in its leaves with respect to their \( x \)-coordinate in increasing order. Let \( H \) be the height of tree \( T \). We denote by \( T_v \) the subtree of \( T \) with root the internal node \( v \).

Let \( P_\ell \) be the root-to-leaf path for leaf \( \ell \) of \( T \). We denote by \( P_\ell^\tau \) the subpath of \( P_\ell \) consisting of nodes with depth \( \geq \tau \). Similarly, \( P_\ell^{\tau, \text{left}} \) \((P_\ell^{\tau, \text{right}})\) denotes the set of nodes that are left (right) children of nodes of \( P_\ell^\tau \) and do not belong
to \( P^T_\ell \). Let \( q = (q_x, q_y) \) be the point stored in leaf \( \ell \) of the tree where \( q_x \) is its \( x \)-coordinate and \( q_y \) is its \( y \)-coordinate. \( P_q \) denotes the search path for \( q_x \), i.e., it is the path from the root to \( \ell \) and it is equal to \( P_\ell \). We augment \( T \) as follows:

- Each internal node \( v \) stores a point \( q_v \), which is the point with the maximum \( y \)-coordinate among all points in its subtree \( T_v \). It also stores its depth.

- Each internal node \( v \) has a secondary data structure \( S_v \), which stores all points in \( T_v \) with respect to \( y \)-coordinate in increasing order. \( S_v \) is implemented with an IS-tree as well as with an h-RMQ structure, where the maximum is w.r.t. the \( x \)-coordinate.

- Each leaf \( \ell \) stores arrays \( L^T_\ell \) and \( R^T_\ell \), where \( 0 \leq \tau \leq H - 1 \), corresponding to sets \( P^T_{\ell, \text{left}} \) and \( P^T_{\ell, \text{right}} \) respectively. More specifically, they contain the points \( q_v \) for each node \( v \) in the corresponding sets. These are sorted with respect to their \( y \)-coordinate and they are implemented with \( q^* \)-heaps and h-RMQ structures, where the maximum is w.r.t. the \( x \)-coordinate.

We also use an IS-tree \( T' \) to allow for efficient predecessor/successor queries in the leaves of \( T \). Finally, tree \( T \) is preprocessed in order to support Lowest Common Ancestor queries in \( O(1) \) time. Since \( T \) is static, one can use the methods of \([48, 50]\) to find the LCA (as well as its depth) of two leaves in \( O(1) \) time by attaching to each node of \( T \) a label. We now move on to the description of the skyline query for a query range \( q = [a, b] \times [d, +\infty) \):

1. We use the IS-tree \( T' \) to find the two leaves \( \ell_a \) and \( \ell_b \) of \( T \) for the search paths \( P_a \) and \( P_b \) respectively. Let \( w \) be the LCA of leaves \( \ell_a \) and \( \ell_b \) and let \( \tau \) be its depth.

2. The successor of \( d \) is located in \( R^T(\ell_a) \) and \( L^T(\ell_b) \) and let these successors be at positions \( \text{succ}_R[d] \) and \( \text{succ}_L[d] \) respectively. In addition, let \( v_1 \) be the node that has the following property: the \( y \)-coordinate of point \( q_{v_1} \) belongs in the range \([d, +\infty)\) and it has the largest \( x \)-coordinate (the \( x \)-coordinate of \( q_{v_1} \) falls in the \([a, b]\) range because of Step 1) among all nodes in \( P^T_{\ell_a, \text{right}} \) and \( P^T_{\ell_b, \text{left}} \).

3. By executing an h-RMQ in \( L^T_{\ell_a} \) and \( R^T_{\ell_a} \) arrays for the range \([\text{succ}_L[d], \tau]\) and \([\text{succ}_R[d], \tau]\) node \( v_1 \) is located. The subtree \( T_{v_1} \) stores the point with the maximum \( x \)-coordinate among all points in the query range \([a, b] \times [d, +\infty)\). By executing a successor query for \( d \) in \( S_{v_1} \) returning the result \( \text{succ}_S[d] \), and then making an h-RMQ in \( S_{v_1} \) for the range \([\text{succ}_S[d], |S_{v_1}|]\), we find and report the required point with the maximum \( x \)-coordinate \( z = (z_x, z_y) \) that belongs to the skyline.
4. The query range now becomes \( q = [a, z_x] \times [z_y, +\infty) \) and we repeat the previous steps until \( S \cap q = \emptyset \).

For each skyline point, we execute \( O(1) \) successor queries in total. Since the \( q^* \)-heap queries and all other steps can be carried out in \( O(1) \) time, the total time cost of the query algorithm is \( O(t \cdot t_{IS}(N)) \) where \( t_{IS}(N) \) is the time required by an IS-tree for a successor query and \( t \) is the answer size. The space cost of the ML-tree is dominated by the space used for implementing the \( L_\tau \ell \), \( R_\tau \ell \) and \( S_v \) sets, which is \( O(N \log^2 N) \) since each point is stored in \( O(\log N) \) \( S_v \) structures and each leaf \( \ell \) among the \( N \) leaves in total, stores \( O(\log N) \) \( L_\tau \ell \) and \( R_\tau \ell \) sequences each of which has size \( O(\log N) \).

4.2.2 The Main Memory Static Linear-Space MLR-tree

We can reduce the space of the data structure described in Section 4.2.1 by using pruning techniques as in [41, 91]. However, pruning alone does not reduce the space to linear. We can obtain a better space complexity by recursive pruning until reaching a tree of constant size, but it will still be superlinear by an iterated logarithm\(^3\) (aggravating by a similar multiplicative term the time complexity of the query). To get an optimal space bound we use a combination of pruning and table lookup, which ends the recursion prematurely.

We reduce the space of the data structure by employing a pruning technique [41, 91] as follows: consider the nodes of \( T \) with height \( 2 \log \log N \). These nodes are the roots of subtrees of \( T \) of size \( O(\log^2 N) \) and there exist \( \Theta \left( \frac{N}{\log^2 N} \right) \) such nodes. Let \( T_1 \) be the tree whose leaves are these nodes and let \( T_2^i \) be the subtrees of these nodes for \( 1 \leq i \leq \Theta \left( \frac{N}{\log^2 N} \right) \). We call \( T_1 \) the first layer of the structure and the subtrees \( T_2^i \) the second layer.

\( T_1 \) and each subtree \( T_2^i \) is implemented as a static non-linear space ML-tree. The representative of each tree \( T_2^i \) is the point with the maximum \( y \)-coordinate among all points in \( T_2^i \). The leaves of \( T_1 \) contain only the representatives of the respective trees \( T_2^i \). Each tree \( T_2^i \) is further pruned at height \( 2 \log \log \log N \) resulting in trees \( T_3^j \) with \( \Theta(\log^2 \log N) \) elements. Once more, \( T_3^j \) contains the representatives of the third layer trees in a similar way as before. Each tree \( T_3^j \) is structured as a table that stores all possible precomputed solutions. Specifically, each \( T_3^j \) is structured by using a \( q^* \)-heap with respect to the \( x \)-coordinate as well as one with respect to the \( y \)-coordinate. In this way, we can extract the position of the successor in \( T_3^j \) with respect to \( x \) and \( y \) coordinates. What is needed to be computed for \( T_3^j \) is the point with the maximum \( x \)-coordinate that lies within a 3-sided range region. To obtain this, we use precomputation and tabulation for all possible results.

\(^3\)The iterated logarithm, written as \( \log^* N \), is equal to the number of times the logarithm must be iteratively applied on \( N \) before the result is \( \leq 1 \) for the first time.
For the sake of generality, assume that the size of $T_3^j$ is $k$. Let the points in $T_3^j$ be $q_1, q_2, \ldots, q_k$ sorted by $x$-coordinate. Let their rank according to $y$-coordinate be given by the function $\alpha(i), 1 \leq i \leq k$. Apparently, function $\alpha$ may generate all possible $k!$ permutations of the $k$ points. We make a four-dimensional table $\text{ANS}$, which is indexed by the number of permutations (one dimension with $k!$ choices) as well as the possible positions of the successor (3 dimensions with $k + 1$ choices for the 3-sided range). Each cell of array $\text{ANS}$ contains the position of the point with the maximum $x$-coordinate for a given permutation that corresponds to a tree $T_3^j$ and the 3-sided range. Each tree $T_3^j$ corresponds to a permutation index that indexes one dimension of table $\text{ANS}$. The other 3 indices are generated by one predecessor and one successor query on the $x$-coordinate and one successor query on the $y$ coordinate. The size of $\text{ANS}$ for each $T_3^j$ is $O(k!(k + 1)^3)$.

We attach a unique label in the range $[1, k!]$ to each one of the $k!$ permutations corresponding to the respective index in array $\text{ANS}$. This label is constructed by enumerating systematically all $k!$ permutations and keeping them in an array of labels. Each label is represented by $O(k \log k)$ bits. Each tree in the third layer is attached with such a label based on the permutation generated by the $y$-coordinates of its points. This is the only step in the building process that requires knowledge of the trees $T_3^j$. Then, we compute for every permutation and for every possible combination of the three (overall) predecessor/successor the rank of the point with the maximum $x$-coordinate.

Although the skyline query changes to incorporate the division of the structure into 3 layers, these changes are not extensive. Let $q = [a, b] \times [d, +\infty)$ be the initial range query. To answer this query on the three layered structure we access the layer 3 trees containing $a$ and $b$ by using the $T'$ tree. Then, we locate the subtrees $T_2^j$ and $T_2^i$ containing the representative leaves of the accessed layer 3 trees. The roots of these subtrees are leaves of $T_1$. The ML query algorithm described in Section 4.2.1 is executed on $T_1$ with these leaves as arguments. Once we reach the node with the maximum $x$-coordinate, we continue in the layer 2 tree corresponding to the representative with the maximum $x$-coordinate located in $T_1$. The same query algorithm is executed on this layer 2 tree and then we move similarly to a tree $T_3^j$ in the third layer.

We make three in total successor queries for $a$, $b$, and $d$ in $T_3^j$ and we use the $\text{ANS}$ table to locate the point with the maximum $x$-coordinate by retrieving the permutation index of $T_3^j$. Let the point $z = (z_x, z_y)$ be the desired point at the third layer. We go back to $T_1$. The range query now becomes $q = [a, z_x] \times [z_y, +\infty)$ and we iterate as described in Section 4.2.1.

The total space required for the data structure depends on the size of each of the three layers. For the first layer, the ML-tree on the $O\left(\frac{N}{\log^2 N}\right)$
Chapter 4. Dynamic 3-sided Skyline Queries

representatives requires \( O\left(\frac{N}{\log^2 N}\log^2\left(\frac{N}{\log^2 N}\right)\right) = O(N) \) space for the leaf structures (all \( P_\ell \) structures for each leaf \( \ell \) are structured as \( q^* \)-heaps and h-RMQ structures requiring linear space). For the \( S_0 \) structures, the total space needed is \( O\left(\frac{N}{\log^2 N}\log N\right) = O(N) \). A similar reasoning can be made for the second layer that consists of \( O\left(\frac{N}{\log^2 N}\right) \) trees with \( O\left(\frac{\log N}{\log\log N}\right)^2 \) representative points of the third layer each for a total space of \( O(\log^2 N) \). In the third layer, we use linear space for the two predecessor data structures (\( q^* \)-heaps) as well as a table of size \( O((4\log^2 \log N)!(4\log^2 \log N + 1)^3) \), which is \( O(N) \). The construction time of the data structure can be similarly derived taking into account that the ANS table can be constructed in \( O(N) \) time. The query time is bounded by the \( O(1) \) number of successor queries per actual resulting skyline point. The following lemma summarizes this discussion and it will be used to design the dynamic data structure.

**Lemma 5.** Given a set of 2-d \( N \) points, we can store them in a static main memory data structure that can be constructed in \( O(N\log N) \) time using \( O(N) \) space. It supports skyline queries in a 3-sided range in \( O(t \cdot t_{IS}(N)) \) worst-case time, where \( t \) is the answer size and \( t_{IS}(N) \) is the time required by an IS-tree for a predecessor/successor query.

### 4.3 The Dynamic MLR-tree

Making the ML-tree described in Section 4.2.2 dynamic involves all layers. The following issues must be tackled in order to make the ML-tree dynamic:

1. Use of a dynamic tree structure with care to how rebalancing operations are performed.

2. The layer 3 trees must have variable size within a predefined range, rebuilding them appropriately as soon as they violate this bound (by splitting or merging/sharing with adjacent trees) - similarly, the permutation index must be appropriately defined in order to allow for variable length permutations.

3. All arrays attached to nodes or leaves must be updated efficiently.

We use global rebuilding [92] to maintain the structure. In particular, let \( N_0 \) be the number of elements stored at the time of the latest reconstruction. At the time when the number of updates exceeds \( rN_0 \), where \( 0 < r < 1 \) is a constant, the whole data structure is reconstructed taking into account that the number of elements is \( rN_0 \). In this way, it is guaranteed that the current number of elements \( N \) is always within the range \( [(1 - r)N_0, (1 + r)N_0] \). We call the time between two successive reconstructions an *epoch*. The
4.3. The Dynamic MLR-tree

tree structure used for the first two layers is a weight-balanced tree, like the $BB[a]$-trees [88].

Henceforth, assume for brevity that $k = \log^2 \log N$. We impose that all trees at layer 3 will have size within the range $[k/4, k]$. To compute the permutation index, if the size of the layer 3 tree is $< k$, then we pad the increasing sequence of elements in the tree with $+\infty$ values in order to have exactly size $k$ (alternatively, we could also count the number of subsets of size in the range $[k/4, k]$ increasing the size of the table ANS but not exceeding the $O(N)$ bound).

Assume that an update operation takes place. The following discussion concerns the case of inserting a new point $q = (q_x, q_y)$ since the case of deleting an existing point $q$ from the structure is symmetric. First, $T'$ is used to locate the predecessor of $q_x$, and in particular to locate the tree $T_{j3}$ of layer 3 that contains the predecessor of $q_x$. Then, $T'$ is updated accordingly. The predecessor of $q_x$ in $T_{j3}$ is located by using the respective $q^*$-heap. If $|T_{j3}| \in [k/4, k]$, then $q_x$ and $q_y$ are inserted in the respective $q^*$-heaps and a new permutation index is computed for $T_{j3}$. If $|T_{j3}| > k$, then $T_{j3}$ is split into two trees with size approximately $k/2$. This means that 4 new $q^*$-heaps must be constructed while two new permutation indices must be computed for the two new trees.

Let $T_2$ be the layer 2 tree that gets the new leaf. Note that $T_2$ is affected either structurally, when one of its leaves $\ell$ at layer 3 splits as in this case ($\ell$ is $T_{j3}$) or it is affected without structural changes, when $q_y$ is maximum among all the $y$-coordinates of $T_{j3}$ and thus the representative of $T_{j3}$ changes. In the latter case, all structures $S_v$ on the path $P_\ell$ of $T_2$ must be updated with the new point. In addition, let $v$ be the highest node with height $h_v$ in $T_2$ that has $p_v = q$ (the point with the maximum $y$-coordinate in its subtree changes to $q$). Then, for all leaves $\ell$ in the subtree of the father of $v$, the $q^*$-heaps for $L_\ell^T$ and $R_\ell^T$ as well as the h-RMQ structures that contain $v$ will be updated, given that $T_3 \geq h_v$. In the former case, we make rebalancing operations on the internal nodes of $T_2$ on the path $P_\ell$. These rebalancing operations result in changing, as in the previous case, the $q^*$-heaps for the $L_\ell^T$ and $R_\ell^T$ while the respective $S_v$ structures of the node $v$ that is rebalanced have to be recomputed as well. Similar changes happen to the tree $T_1$ of the first layer given that either a tree of the second layer splits or its maximum element is updated. In case of deleting $x$, the 3 layers of the ML-tree are handled similarly.

Recall that the time complexity of the update operation supported by the IS-tree and $q^*$-heap is $O(1)$. The change of the point with the maximum $y$-coordinate can always propagate from $T_{j3}$ to the root of $T_1$. $T_{j3}$ can be updated in $O(|T_{j3}|)$ time since the two updates in $q^*$-heaps cost $O(1)$ while the computation of the permutation index costs $O(|T_{j3}|)$. Let the respective tree in the second layer be $T_2$. Then, the cost for changing the point with the maximum
Chapter 4. Dynamic 3-sided Skyline Queries

$y$-coordinate in each node on the path from the leaf to the root of $T_2$ is related to the update cost for the $L^T$ and $R^T$ lists as well as for the $S_v$ structures. In particular, all $O(|T_2| \log |T_2|)$ lists $L^T$ and $R^T$ are updated (deletion of the previous point and insertion of the new one in a $q'$-heap) in $O(|T_2| \log |T_2|)$ time. Similarly, a deletion and an insertion is carried out in each $S_v$ structure in $O(\log |T_2|)$ total time. The same holds for the tree $T_1$ netting a total complexity of $O(|T_1| \log |T_1|)$.

Rebalancing operations on the level 2 trees as well as on the level 1 tree of the structure may be applied when splits or fusions of leaves of level 2 trees take place. Since level 2 trees are exponentially smaller than the level 1 tree, the cost is dominated by the rebalancing operations at $T_1$. Assume an update operation at a leaf $\ell$ of $T_1$. In the worst case, each $S_v$ structure may have to be rebuilt and similarly to the previous paragraph the $L^T$ and $R^T$ structures need to be updated. The total cost is equal to $O(|T_1| \log^2 |T_1|)$ for the $O(|T_1| \log |T_1|)$ lists while it is $O(|T_1|)$ for the $S_v$ structures since the reconstruction of the $S_v$ structure of the root $r$ dominates the cost. One can similarly reason for level 2 trees. However, the amortized cost is way lower for two reasons: 1. A leaf of $T_1$ is updated roughly every $O(\log^2 N)$ update operations and 2. The weight property of the tree structures guarantees that costly operations are rare. By using a standard weight property argument along with the above two reasons we get that the amortized rebalancing cost is $O \left( \log^2 N + \frac{|T_1| \log N}{N} \right)$. This amortized cost is dominated by the cost to update the maximum element, in which case the worst-case as well as the amortized case coincide. Thus, we obtain the following theorem:

**Theorem 6.** Given a set of $N$ points we can store them in a dynamic main memory data structure that uses $O(N)$ space and supports update operations in $O \left( \frac{N}{\log N} \right)$ time in the worst case. It supports skyline queries in a 3-sided range in $O(t \cdot t_{IS}(N))$ worst-case time, where $t$ is the answer size and $t_{IS}(N)$ is the time required by an IS-tree for a predecessor query.

Although rebalancing operations are efficient in an amortized sense, the change of maximum depends on the user and in the worst-case this change can propagate to the root in each update operation. In the following, we overcome this problem by making a rather strong assumption about the distribution of the points.

### 4.3.1 Exploiting the Distribution of the Elements

To reduce the huge worst-case update cost of Theorem 6 we have to tackle the propagation of maximum elements. To accomplish this we assume that the coordinates of the points are generated by discrete distributions. The result can also be attained by minor modifications for the case of continuous distributions. Assume that a new point $q = (q_x, q_y)$ is to be inserted in the ML-tree.
4.3. The Dynamic MLR-tree

Let \( q \) be stored in level 2 tree \( T_2 \) based on \( q_x \). We call the point \( q \) violating if \( q_y \) is the maximum \( y \)-coordinate among all \( y \)-coordinates of the points in \( T_2 \). When a new point is violating it means that a costly update operation must be performed on \( T_1 \). In the following, we show that under assumptions on the generating distributions of the \( x \) and \( y \) coordinates of points we can prove that during an epoch only \( O(\log N) \) violations will happen.

We assume that all points have their \( x \)-coordinate generated by the same discrete distribution \( \mu \) that is \( (f_1(N) = \frac{N}{(\log \log N)^{1+\epsilon}}, f_2(N) = N^{1-\delta}) \)-smooth, where \( \epsilon > 0 \) and \( \delta \in (0, 1) \) are constants. We also assume that the \( y \)-coordinates of all points are generated by a restricted set of discrete distributions \( \mathcal{Y} \), on the sample space \( \{y_1, y_2, \ldots \} \) such that \( y_i < y_{i+1}, \forall i \geq 1 \). In particular, let an arbitrary point \( p = (p_x, p_y) \) and let \( \alpha = Pr[p_y > y_1] \). A distribution belongs in the family of distributions \( \mathcal{Y} \) if \( \alpha \leq \left( \frac{\log N}{N} \right)^{\frac{1}{\log N}} \) which tends to \( e^{-1} \) as \( n \to +\infty \). The family of distributions \( \mathcal{Y} \) contain among others the Power Law and the Zipfian distributions. Finally, we assume that deletions are equiprobable for each existing point in the structure. In a nutshell, the structure requires that during an epoch tree \( T_1 \) remains intact and only level 2 and level 3 trees are updated.

The construction of the static tree \( T_1 \) now follows the lines of [62]. Assume that the \( x \)-coordinates are in the range \([x_1, x_2] \). Then, this range is recursively divided into \( f_1(N) \) subranges. The terminating condition for the recursion is when a subrange has \( \leq \log^2 N \) elements. Note that the bounds of these subranges only depend on the properties of the distribution. This construction is necessary to ensure certain probabilistic properties for discrete distributions. However, instead of building an interpolation search tree, we build a binary tree on these subranges and then continue building the lists of the leaves and the internal nodes as in the previous structures. The elements within each subrange correspond to a level 2 tree whose leaves are level 3 trees. The following theorem regarding each epoch is obtained from [62]:

**Theorem 7.** The construction of the terminating subranges defining the level 2 trees can be performed in \( O(N) \) time in expectation with high probability. Each level 2 tree has \( \Theta(\log^2 N) \) points in expectation with high probability during an epoch.

The above theorem guarantees that the size of the buckets is not expected to change considerably and as a result we are allowed to assume that no update operations will happen on \( T_1 \). This is the result of assuming that the \( x \)-coordinates of the points inserted are generated by an \( \left( \frac{N}{(\log \log N)^{1+\epsilon}}, N^{1-\delta} \right) \)-smooth distribution.

The reduction of the number of violating points during an epoch is attributed to our assumption that the \( y \) coordinates follow a distribution that

\[ \text{Probability that the } y \text{ coordinate is strictly larger than the minimum element in the sample space } \{ y_1, y_2, \ldots \} \]
belongs to the \( \mathcal{Y} \) family of distributions. All violating points are stored explicitly and since there are only a few in expectation during an epoch, we can easily support the query operation. After the end of the epoch, the new structure has no violating points stored explicitly. The following theorem from \[64\] guarantees the small number of violating points during an epoch:

**Theorem 8.** For a sequence of \( \Theta(n) \) updates, the expected number of violations is \( O(\log n) \), assuming that the \( x \) coordinates are drawn from an \( (N/((\log \log N))^{1+\epsilon}, N^{1-\delta}) \)-smooth distribution, where \( \epsilon > 0 \) and \( \delta \in (0,1) \) are constants, and the \( y \) coordinates are drawn from the restricted class of distributions \( \mathcal{Y} \) with sample space \( \{y_1, y_2, \ldots\} \), where \( y_i < y_{i+1}, \forall i \geq 1 \), such that it holds that \( \alpha \leq \left( \frac{\log N}{N} \right)^{\frac{1}{\log N}} \to e^{-1} \), where \( \alpha = \Pr[p_y > y_1] \) for an arbitrary point \( p = (p_x, p_y) \).

The theorem that describes the result attained in this chapter for 3-sided dynamic skyline queries follows:

**Theorem 9.** Given a set of \( N \) 2-d points, whose \( x \) coordinates are generated by an \( (N/((\log \log N))^{1+\epsilon}, N^{1-\delta}) \)-smooth distribution, where \( \epsilon > 0 \) and \( \delta \in (0,1) \) are constants, and the \( y \) coordinates are drawn from the restricted class of distributions \( \mathcal{Y} \), we can store them in a dynamic main memory data structure that uses \( O(N) \) space and supports update operations in \( O(\log^2 N \log \log N) \) expected time with high probability. It supports skyline queries in a 3-sided range in \( O(t \log \log N) \) worst-case time, where \( t \) is the answer size.
Chapter 5

HiNode: An Asymptotically Space-Optimal Storage Model for Historical Queries on Graphs

Moving on to the second part of the thesis, the attention shifts to issues that revolve around efficient historical information management in the presence of massive graph datasets. In this chapter, the asymptotically space-optimal prototype system HiNode is presented, which follows an entity-centric design approach and avoids the use of snapshot reconstructing mechanisms that feature deltas. More specifically, the system advocates a vertex-centric maintenance of history, that is, the changes on the graph are represented and stored as time intervals on the affected vertices. This fact, favors computation on a more “local” region since only the sub-graphs that are related to a query have to be reconstructed.

The principal goal is to show that HiNode, as a pure vertex-centric storage model, exhibits great potential. This is showcased both theoretically through a qualitative system comparison with two main competitors ($G^*$ [76, 107] and TGI [66]) as well as experimentally using the $G^*$ parallel graph processing system which is available for download. To accomplish this goal, the work focused on the space usage of the storage model as well as its functionality. One of the goals is to show that in this pure vertex-centric approach the space overhead is minimal leading to indirect time savings in queries as well.

In addition, HiNode can support various sets of queries focusing largely on local queries where the pure vertex-centric approach is naturally suitable and shows great potential. This is indicated by theoretical as well as experimental results. Finally, experimental and theoretical indications are provided that HiNode can support general queries very well. In summary, the following contributions are made:

- Implementation of the first purely vertex-centric approach to the organization of evolving graphs. The approach is especially suitable for local queries that involve one or more graph snapshots.

- Demonstration of HiNode’s space efficiency: The system’s storing and indexing approach is more space efficient than existing approaches.
• Establishment of a set of primitive operators on top of which many complex historic graph execution plans can be built. Additionally, evidence is provided that these operations can be efficiently supported with results comparable or even better than those of other time-centric approaches.

• The HiNode system can efficiently support more general notions of time with respect to the evolution of the graph. The evolution of the graph may be represented in a linear or a tree-like fashion whereas all solutions up to this point support a linear notion of time.

• HiNode’s approach handles in a very natural way incremental updates that generate new snapshots. The granularity of evolutions can be arbitrarily fine whereas all other solutions try to overcome this problem by either employing logging or by grouping together many updates which naturally leads to handling a rather small number of snapshots.

• To prove the practicality of HiNode’s approach, it was implemented and incorporated in the $G^*$ parallel graph processing system [76, 107]. Experimental evaluation shows that, due to the low cost of accessing stored data, the reconstruction overheads are outweighed in most of the cases, and HiNode is efficient even for queries that require the reconstruction of complete snapshots.

The rest of this chapter is organized as follows. Section 5.1, presents the main definitions and notation. The storage model of HiNode is introduced in Section 5.2. Core algorithms and practical considerations are presented in Section 5.3. Section 5.4, showcases the efficiency of HiNode through experimental evaluation. Finally, Section 5.5 discusses multiple universes and efficiently registering updates.

5.1 Preliminaries

We follow a linear notion of time and define it to be a strictly increasing quantity measured in indivisible time intervals. We discuss a tree-like multiple history in Section 5.5. Let $\mathcal{G} = \{G_1, G_2, G_3, \ldots\}$ be the sequence of graph snapshots to be stored and accessed. The sequence does not have a final snapshot, in the sense that it is constantly evolving. For a graph $G_i \in \mathcal{G}$, the snapshot $G_i = (V_i, E_i)$ corresponds to the graph $G$ at time instance $i$ and is characterized by a set of vertices $V_i$ and a set of incoming and outgoing edges $E_i$, with each vertex or edge possessing its own set of (possibly multi-valued) attributes. Conceptually, one may obtain the snapshot of $G_i$ by applying a set of insertions, deletions or value updates to the vertices and edges of $G_{i-1}$. Moreover, since our theoretical model works with valid time, a user is able to update a specific snapshot even though it may not currently be the last in the
5.2 Storage Model

We propose a storage model for the graph sequence \( \mathcal{G} \). This storage model supports a variety of fundamental access and update operations that allow the user to access any graph \( G_i \) (either the whole graph or a particular subgraph) as well as alter the graphs at any time instance. Furthermore, the storage model does not rely on there being no changes in the schema of the vertices and supports different attributes in the same vertex over time. In the following, we begin with an overview of the proposed data structure, which comprises nested elements at different levels, followed by a description of the supported operations. We conclude the section by providing an asymptotic analysis of the space and time complexities.
### Chapter 5 Notation Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G_i$</td>
<td>A snapshot of graph $G$ at time instance $i$</td>
</tr>
<tr>
<td>$V_i$</td>
<td>The set of vertices of $G_i$</td>
</tr>
<tr>
<td>$E_i$</td>
<td>The set of incoming and outgoing edges of $G_i$</td>
</tr>
<tr>
<td>$\mathcal{G}$</td>
<td>A sequence of $G_i$ for various $i$</td>
</tr>
<tr>
<td>$\mathcal{V}$</td>
<td>The set of all vertices in $\mathcal{G}$</td>
</tr>
<tr>
<td>$\mathcal{E}$</td>
<td>The set of all edges in $\mathcal{G}$</td>
</tr>
<tr>
<td>$</td>
<td>v</td>
</tr>
<tr>
<td>$</td>
<td>v_t</td>
</tr>
<tr>
<td>$\mathcal{S}$</td>
<td>Total size of all vertices in $V_i$</td>
</tr>
<tr>
<td>$m$</td>
<td>The total count of changes (insertions, deletions, updates) made from the first snapshot to the (currently) last snapshot in $\mathcal{G}$</td>
</tr>
<tr>
<td>$I$</td>
<td>External interval tree that maintains the “lifetime” of each vertex in $\mathcal{V}$</td>
</tr>
<tr>
<td>$T^v_{t_s,t_e}$</td>
<td>An interval in $I$ signifying that vertex $v$ in $\mathcal{G}$ is active between the time instances $t_s$ and $t_e$</td>
</tr>
<tr>
<td>$D_v$</td>
<td>Diachronic node of vertex $v$</td>
</tr>
<tr>
<td>$f$</td>
<td>An identifier of a particular attribute (e.g. name, weight etc.) or edge</td>
</tr>
<tr>
<td>$I_v$</td>
<td>External interval tree of $D_v$ that maintains information regarding all the attributes of $v$</td>
</tr>
<tr>
<td>$A^f_v$</td>
<td>A B-tree for the attribute with identifier $f$ of vertex $v$</td>
</tr>
<tr>
<td>$B_v$</td>
<td>A B-tree used as an index over all $A^f_v$ trees</td>
</tr>
<tr>
<td>$B$</td>
<td>A B-tree used as an index over the identifiers of all the diachronic nodes</td>
</tr>
<tr>
<td>$\mathcal{B}$</td>
<td>Disk block size</td>
</tr>
</tbody>
</table>

**Table 5.1: Chapter 5 Notation Table**

### 5.2.1 Data Structure Overview

In this section, we will provide an overview of the data structure and we will use a data structure built on the evolving graph sequence of Figure 5.1 as a running example (Figure 5.3). Recall that $\mathcal{G} = \{G_1, G_2, G_3, \ldots\}$ is a sequence of graph snapshots with each $G_i \in \mathcal{G}$ corresponding to a snapshot of the graph $G$ at time instance $i$. A vertex $v \in G_i$ is characterized by a set of attributes (e.g. color), a set of incoming edges from the other vertices of $G_i$ and a set of outgoing edges to the other vertices of $G_i$. We construct an external interval tree $I$ that maintains a set of intervals $\{T^v_{t_s,t_e}\}$ where an interval $T^v_{t_s,t_e}$ signifies the “lifetime” of a vertex $v$, i.e. from time instance $t_s$ to time instance $t_e$. We mark a vertex to be “active” (alive) up until the
5.2. Storage Model

latest time instance, by setting the \( t_e \) value to be \( +\infty \). As an example, in Figure 5.3 we can see that at time instance 3, interval tree \( I \) stores four intervals, namely \( \{ T_{3, \infty}^a, T_{3, \infty}^b, T_{3, \infty}^c, T_{3, \infty}^d \} \). Finally, each interval \( T_{v,t}^e \) is augmented with a pointer (handle) to an additional data structure for each vertex \( v \), called diachronic node.

A diachronic node \( D_v \) of a vertex \( v \) maintains a collection of data structures corresponding to the full vertex history in the sequence \( G \), i.e. when that vertex was inserted, all corresponding changes to its edges or attributes and finally its deletion time (if applicable). More formally, a diachronic node \( D_v \) maintains an external interval tree \( I_v \) which stores information regarding all of \( v \)'s characteristics (attributes and edges) throughout the entire \( G \) sequence. An interval in \( I_v \) is stored as a quadruple \((f, \{\ell_1, \ell_2, \ldots\}, t_s, t_e)\), where \( f \) is the identifier of the attribute that has values \( \ell_1, \ell_2, \ldots \) during the time interval \([t_s, t_e]\). Note that an edge belonging to \( v \) (i.e. one endpoint of the edge is \( v \)), can be represented as an attribute of \( v \) by using one value \( \ell_i \) to denote the other end of the edge, another value \( \ell_j \) to mark the edge as incoming or outgoing and a last value \( \ell_h \) that is used as a handle to the diachronic node corresponding to the vertex in the other end of the edge. The remaining \( \ell \) values can be used to store the attributes of the edge themselves (e.g. weight). In the running example of Figure 5.3, we can see that at time instance 3, the interval tree \( I_a \) of the diachronic node \( D_a \) stores four intervals: one for the label of the vertex and three for all the edges that have occurred thus far for vertex \( a \).

Additionally, the diachronic node maintains a B-Tree for each attribute and for each individual edge of the vertex. The B-tree corresponding to the attribute with identifier \( f \) of vertex \( v \) is denoted as \( A_v^f \) and is used to maintain the entire history of changes of \( f \) between the different snapshots in \( G \). Each record in \( A_v^f \) is a triple \((\{\ell_1, \ell_2, \ldots\}, t_s, t_e)\) where \( \ell_1, \ell_2, \ldots \) are the values of \( f \) during the time interval \([t_s, t_e]\). The edges of \( v \) are represented in a similar manner to attributes taking into account the values \( \ell_i, \ell_j \) and \( \ell_h \) which were discussed on the previous paragraph. Using the \( A_v^f \) trees, which are built on the \([t_s, t_e]\) intervals, we can support stabbing queries for a particular set of attributes within the diachronic node, without aggravating asymptotically the space usage. Given the fact that the count of \( A_v^f \) trees is dependant on the edge count of \( v \) and to facilitate efficient searching of a specific \( A_v^f \) tree, we maintain a B-Tree \( B_v \) over all \( A_v^f \) trees. Finally, we maintain the location of all diachronic nodes using a B-Tree dictionary \( B \) built on the IDs of the diachronic nodes. With regard to the example of Figure 5.3, at time instance 3 the diachronic node of vertex \( a \) has four \( A_v^f \) B-trees each containing one record.

Figure 5.2 shows the proposed data structure, where \( |v| \) is the size of \( v \) (i.e. the count of attributes and edges of \( v \) across all of \( G \)). The full arrows are
handles to diachronic nodes from the intervals in $\mathcal{I}$ while the dashed arrows signify handles to diachronic nodes from $\mathcal{B}$. Depending on the operation we may use either option to locate a specific diachronic node. The full example of the data structure built on the evolving graph sequence of Figure 5.1 can be seen in Figure 5.3. We name our storage model as HiNode standing for “History in the (diachronic) Node”.

### 5.2.2 Basic Operations

We implement a basic set of operations over the graph sequence that can be used as primitive for implementing more complex operations concerning a subgraph (or the entire graph) at a particular time instance. Henceforth, we use the terms fields and attributes interchangeably to refer to the attributes of each vertex and assume that all operations refer to a vertex in the sequence labeled as $v$. Each operation description is accompanied by pseudocode.

**InsertVertex** (Algorithm 1) creates an interval $\mathcal{T}_{v_{ts,t_e}}$ that corresponds to a new vertex $v$ in the sequence $\mathcal{G}$ that is inserted at time $t_s$. Furthermore, an empty diachronic node $\mathcal{D}_v$ is created for $v$ and a pointer $p_v$ to $\mathcal{D}_v$ is attached to $\mathcal{T}_{v_{ts,t_e}}$ and inserted in $\mathcal{B}$. Finally, the interval $\mathcal{T}_{v_{ts,t_e}}$ is inserted in $\mathcal{I}$ and the $p_v$ pointer is returned. If at the time of insertion, the end time $t_e$ is not known, we set it to infinity. Note that even if $v$ gets deleted at some time instance $t$, we keep $\mathcal{D}_v$ to facilitate historical queries.
Figure 5.3: The data structure built on the graph sequence of Figure 5.1 as seen at time instance 3.

ReadAttribute (Algorithm 2) returns the set of values \{\ell_1, \ell_2, \ldots\} of the field \(f\) in vertex \(v\) at time \(t\). To realize this operation we retrieve the diachronic node \(D_v\) by querying \(B\) for \(\text{id} v\). Afterwards, we obtain the \(A^f_v\) tree using \(B_v\) and perform a query for time \(t\). If there exists a set of values \(S = \{\ell_1, \ell_2, \ldots\}\) for \(f\) at the time instance \(t\) it is returned, otherwise the operation returns a NULL value.

WriteAttribute (Algorithm 3) assigns a set of values \(S = \{\ell_1, \ell_2, \ldots\}\) which are valid for the time interval \([t_s, t_e]\) to the field \(f\) of vertex \(v\). To achieve this, firstly we obtain the diachronic node \(D_v\) of \(v\) by searching \(B\) for \(\text{id} v\). Following that, we perform a stabbing query on the respective \(A^f_v\) tree of \(f\) for each of the \(t_s\) and \(t_e\) endpoints. If the stabbing queries do not return any interval then the field \(f\) does not have any values associated with it in the time interval \([t_s, t_e]\) and we insert the relevant data in \(I_v\) and \(A^f_v\) directly; otherwise, we retrieve the (at most two) returned interval(s) (e.g. \([t'_s, t'_e]\) and...
In the first case, the field $f$ does not have any values associated with it in the time interval $[t_s, t_e]$. In that case we proceed as follows: We insert a quadruple in $I_v$ of the form $(f, \{\ell_1, \ell_2, \ldots\}, t_s, t_e)$. In addition, a record $(\{\ell_1, \ell_2, \ldots\}, t_s, t_e)$ is stored in $f$’s respective B-tree $A_v^f$.

In the second case, the field $f$ has values associated with it in the time interval $[t_s, t_e]$, i.e. there exist (up to) two intervals $[t'_s, t'_e]$ and $[t''_s, t''_e]$ in the data structure, such that either (a) $t'_s < t_s < t'_e < t_e$, (b) $t_s < t'_s < t_e < t'_e$, (c) $t'_s < t_s < t_e < t'_e$ or (d) $t'_s < t_s < (t'_e = t''_e) < t_e < t''_e$ is true (Figure 5.4). In that case, we search $I_v$ for $[t'_s, t'_e]$ corresponding to the field $f$ (and $[t''_s, t''_e]$ if it exists) by simulating an insertion of this interval in $I_v$. Let $v'$ be the node of $[t''_s, t''_e]$, remove their portion that is being overlapped by $[t_s, t_e]$ and finally insert the relevant data in $I_v$ and $A_v^f$. The two cases are described in detail below.

**Algorithm 1** InsertVertex($v, t_s, t_e$)

**Input:** vertex id $v$, start time $t_s$, end time $t_e$

**Output:** a pointer $p_v$ to the diachronic node of $v$

1: $T^v_{t_s, t_e} \leftarrow$ **new** interval $(v, t_s, t_e)$
2: $p_v \leftarrow $ pointer(\textbf{new} diachronic node($v$))
3: $T^v_{t_s, t_e}$.attach($p_v$)
4: $B$.insert($p_v$) \>	extit{$p_v$ is inserted in $B$ along with the ID of $v$}
5: $I$.insert($T^v_{t_s, t_e}$)
6: return $p_v$

**Algorithm 2** ReadAttribute($v, f, t$)

**Input:** vertex id $v$, attribute or field $f$, time instance $t$

**Output:** a set of values $S = \{\ell_1, \ell_2, \ldots\}$ corresponding to the values of $f$ at time $t$; NULL otherwise

1: $D_v \leftarrow B_v$.query($v$)
2: $A_v^f \leftarrow D_v.B_v$.query($f$)
3: $S \leftarrow A_v^f$.stab($t$)
4: if $S \neq \emptyset$ then return $S$
5: else return NULL
6: end if

![Figure 5.4: Cases of existing intervals for the field $f$](image-url)
5.2. Storage Model

\[ \mathcal{I} \] that interval \([t'_s, t'_e]\) is to be stored. After locating the at most three lists in which it is to be stored we search these lists based on the endpoints of \([t'_s, t'_e]\). If there are more than one such intervals then we use the identifier of \([t'_s, t'_e]\) to search among them and locate this interval. The same procedure is applied for \([t''_s, t''_e]\).

Afterwards, we perform a series of interval insertions and deletions in \(\mathcal{I}\) and the corresponding \(A^I_v\) B-tree depending on the subcases presented below (the resulting intervals end up with the appropriate set of values based on their original intervals):

**Subcase (a)** Deletion of \([t'_s, t'_e]\) followed by the insertion of \([t'_s, t_s], [t_s, t'_e] \) and \([t'_e, t_e]\)

**Subcase (b)** Deletion of \([t'_s, t'_e]\) followed by the insertion of \([t_s, t'_s], [t'_s, t_e] \) and \([t_e, t'_e]\)

**Subcase (c)** Deletion of \([t'_s, t'_e]\) followed by the insertion of \([t'_s, t_s], [t_s, t_e] \) and \([t_e, t'_e]\)

**Subcase (d)** Deletion of \([t'_s, t'_e]\) and \([t''_s, t''_e]\) followed by the insertion of \([t'_s, t_s], [t'_s, t'_e] \), \([t'_s, t'_e], [t'_e, t_e] \) and \([t_e, t''_e]\)

Finally, the \texttt{WriteAttribute} operation is also used in a similar manner to delete a particular interval by specifying the field \(f\) and the interval \([t_s, t_e]\) to be deleted and passing a \texttt{NULL} value as the set of values. Note that the deletion operation is meaningless in the transaction time setting.

We conclude the operation by pointing out that the first case represents the insertion of data corresponding to a particular vertex’s field, while the second case can be seen as correcting the data that the vertex already had. To that end, we do not permit cases where the insertion of an interval would delete an already existing interval (e.g. \(t_s < t'_s < t'_e < t_e\)) as that would result in the loss of information. In those cases, the user should explicitly delete the related intervals before inserting the new one.

\texttt{ReadVertex} (Algorithm 4) returns a pointer to an object \(u\) that corresponds to the vertex \(v\) as seen at time \(t\). This is realized by obtaining the diachronic node \(\mathcal{D}_v\) of \(v\) using \(\mathcal{B}\) and then performing a stabbing query to \(\mathcal{I} \). The stabbing query returns the set of values at time instance \(t\) for each field \(f\) in \(v\). After following this approach, all the resulting objects are collected and put in an object \(u\) which is returned by \(p_u\).

5.2.3 Analysis of Space and Time Complexities

Our space and time cost analysis is based on the relevant costs of B-Trees and external interval trees. Assuming that the first snapshot in the sequence is
Algorithm 3 WriteAttribute(v, f, {ℓ₁, ℓ₂, . . . }, tₛ, tₑ)

Input: vertex id v, attribute or field f, a set of values {ℓ₁, ℓ₂, . . . }, start time tₛ, end time tₑ

Output: -
1: \( D_v \leftarrow \mathcal{B}.\text{query}(v) \)
2: \( A_v^f \leftarrow D_v.B_v.\text{query}(f) \)
3: \( \text{hasValues} \leftarrow \text{false} \) ▷ Boolean used to distinguish between the two cases
4: if \( A_v^f.\text{stab}(tₛ) \neq \emptyset \) or \( A_v^f.\text{stab}(tₑ) \neq \emptyset \) then
5: Retrieve the interval(s) from \( A_v^f \)
6: \( \text{hasValues} \leftarrow \text{true} \)
7: end if
8: if \( \neg \text{hasValues} \) then ▷ Case 1
9: \( \mathcal{I}_v.\text{insert}((f, \{ℓ₁, ℓ₂, . . . \}, tₛ, tₑ)) \)
10: \( A_v^f.\text{insert}((\{ℓ₁, ℓ₂, . . . \}, tₛ, tₑ)) \)
11: else ▷ Case 2
12: Using the interval(s) retrieved from \( A_v^f \), retrieve the corresponding interval(s) in \( \mathcal{I}_v \)
13: if \( \exists [tₛ'', tₑ''] \) then
14: if \( tₛ' < tₛ < tₑ' < tₑ \) then ▷ Subcase a
15: In \( \mathcal{I}_v, A_v^f: \text{Delete} [tₛ', tₑ']. \text{Insert} [tₛ', tₛ], [tₛ, tₑ'], [tₑ', tₑ] \)
16: else if \( tₛ < tₛ' < tₑ < tₑ' \) then ▷ Subcase b
17: In \( \mathcal{I}_v, A_v^f: \text{Delete} [tₛ', tₑ']. \text{Insert} [tₛ, tₛ'], [tₛ', tₑ], [tₑ, tₑ'] \)
18: else if \( tₛ' < tₛ < tₑ < tₑ' \) then ▷ Subcase c
19: In \( \mathcal{I}_v, A_v^f: \text{Delete} [tₛ', tₑ']. \text{Insert} [tₛ', tₛ], [tₛ, tₑ'], [tₑ, tₑ'] \)
20: end if
21: else ▷ Subcase d
22: In \( \mathcal{I}_v, A_v^f: \text{Delete} [tₛ', tₑ'], [tₛ'', tₑ'']. \text{Insert} [tₛ', tₛ], [tₛ, tₑ'], [tₑ'', tₑ'] \)
23: end if
24: end if

Algorithm 4 ReadVertex(v, t)

Input: vertex id v, time instance t

Output: a pointer \( p_v \) to an object \( u \) that corresponds to \( v \) as seen at \( t \)
1: \( D_v \leftarrow \mathcal{B}.\text{query}(v) \)
2: \( \mathcal{P} \leftarrow D_v.\mathcal{I}_u.\text{stab}(t) \)
3: \( u \leftarrow \text{new vertex}(v, t) \)
4: for each \( f = \text{attribute or edge} \) in \( \mathcal{P} \) do
5: \( u.\text{add}(f) \)
6: end for
7: return pointer(u)

empty, each of the \( m \) changes occurring in the time-evolving sequence is ultimately stored \( O(1) \) times in \( O(1) \) linear-sized data structures. More specifically, in the Insert Vertex operation we insert an interval in \( \mathcal{I} \) and a record
5.2. Storage Model

in $B$ for each newly created vertex while in each WriteAttribute operation we insert up to three intervals and records in $I_v$ and the corresponding $A^f_v$ B-tree respectively. This brings the total space usage for $m$ changes to $O(m/B)$ which is asymptotically optimal with respect to the number of changes.

We analyze the time cost of each operation. The InsertVertex operation requires $O(\log_B m)$ time, since the insertion of $T^v_{t,s,t_e}$ in $I$ and the insertion of the diachronic node pointer in $B$ each require $O(\log_B m)$ time (the creation of the diachronic node itself requires constant time).

The ReadAttribute operation requires $O(\log_B m)$ time in total. Retrieving $D_v$ by querying $B$ and obtaining $A^f_v$ require $O(\log_B m)$ time each. Finally, the query in $A^f_v$ also requires $O(\log_B m)$ time.

We will analyze the WriteAttribute operation by separately analyzing its two cases. Firstly, obtaining $D_v$ is done in $O(\log_B m)$ time. To determine which case stands true, we perform two calls to ReadAttribute that require $O(\log_B m)$ time in total. In the first case we perform two insertions, each requiring $O(\log_B m)$ time.

In the second case, searching for $[t'_s,t'_e]$ (and potentially $[t''_s,t''_e]$) in $I_v$ can be done in $O(\log_B m)$ total time. In any of the resulting subcases we perform a series of $O(1)$ deletions and insertions each requiring $O(\log_B m)$ time. Thus, the second case also requires $O(\log_B m)$ time, yielding $O(\log_B m)$ total time for the operation.

To access a full vertex $v$ at time $t$, we retrieve $D_v$ by querying $B$ and then perform a stabbing query to $I_v$ by using the operation ReadVertex. The total cost is $O(\log_B m + \left|v_t\right|/B)$, where $\left|v_t\right|$ is the size of vertex $v$ at time $t$.

The following theorem summarizes the obtained space and time results.

**Theorem 10.** We can maintain a time-evolving graph sequence in a data structure using $O(m/B)$ space, where $m$ is the total number of changes in the sequence and $B$ is the disk block size. The data structure supports the following basic operations:
1) InsertVertex in $O(\log_B m)$ time,
2) ReadAttribute in $O(\log_B m)$ time,
3) WriteAttribute in $O(\log_B m)$ time,
4) ReadVertex in $O(\log_B m + \left|v_t\right|/B)$ time, where $\left|v_t\right|$ is the size of the vertex $v$ at time $t$ (i.e. the count of attributes and edges of $v$ at time $t$).

5.2.4 Qualitative Comparison

Here we provide a qualitative comparison between different methods. To do that, we assume only transaction time, since not all solutions support valid time, as we and TGI do. This comparison is based on the formal $\Delta$ framework introduced in [66], which is summarized as follows. An ephemeral vertex is a vertex at a specific time instance. Thus, one needs to specify the time instance $t$ for which we are interested to retrieve the ephemeral vertex. The ephemeral vertex contains an identifier, a list of incoming and outgoing edges (a list of
Edge list is a sequence of successive events sorted in a chronological order. Edges that constitute the changes between two successive time instances. An \( \Delta \) required by each respective operation. Required by each model while the rest of the columns refer to the access time between different methods in Table 5.2. Column “Space” refers to the space required by each model while the rest of the columns refer to the access time required by each respective operation.

<table>
<thead>
<tr>
<th>Model</th>
<th>Space</th>
<th>Edge Insertion</th>
<th>Snapshot</th>
<th>Ephemeral Vertex</th>
<th>Versioned Vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPTIMAL</td>
<td>( m )</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG</td>
<td>( m )</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COPY</td>
<td>( m^2 )</td>
<td>( m )</td>
<td>(</td>
<td>S</td>
<td>)</td>
</tr>
<tr>
<td>COPY+LOG</td>
<td>( \frac{m^2}{\lg m} )</td>
<td>( \frac{m}{\lg m} )</td>
<td>(</td>
<td>S</td>
<td>+</td>
</tr>
<tr>
<td>TGI</td>
<td>( \frac{\lg m}{\lg m} )</td>
<td>( \frac{\lg m}{\lg m} )</td>
<td>( h</td>
<td>S</td>
<td>+</td>
</tr>
<tr>
<td>( G^* )</td>
<td>( \frac{m + m^2}{\lg m} )</td>
<td>( \frac{\lg m +</td>
<td>E</td>
<td>+ m}{\lg m} )</td>
<td>( \frac{</td>
</tr>
<tr>
<td>HiNode</td>
<td>( m )</td>
<td>( \lg m + \lg</td>
<td>C</td>
<td>)</td>
<td>(</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>Ephemeral 1-Hop</th>
<th>Versioned 1-Hop</th>
<th>Ephemeral Subgraph</th>
<th>Versioned Subgraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPTIMAL</td>
<td>( d )</td>
<td>( d</td>
<td>C</td>
<td>)</td>
</tr>
<tr>
<td>LOG</td>
<td>(</td>
<td>S</td>
<td>)</td>
<td>( m )</td>
</tr>
<tr>
<td>COPY</td>
<td>(</td>
<td>S</td>
<td>+</td>
<td>E</td>
</tr>
<tr>
<td>COPY+LOG</td>
<td>( h</td>
<td>S</td>
<td>+</td>
<td>E</td>
</tr>
<tr>
<td>TGI</td>
<td>( \frac{</td>
<td>C</td>
<td>m}{\lg m} +</td>
<td>E</td>
</tr>
<tr>
<td>( G^* )</td>
<td>( \frac{</td>
<td>C</td>
<td>m}{\lg m} +</td>
<td>E</td>
</tr>
<tr>
<td>HiNode</td>
<td>( \lg m +</td>
<td>A</td>
<td>+ \lg</td>
<td>C</td>
</tr>
</tbody>
</table>

TABLE 5.2: HiNode Qualitative Comparison. The table is split into two pieces in order to facilitate readability. Comparison of storage models w.r.t. space and access time on various operations. Multiplicative and additive constant factors are discarded. Some of the following parameters either correspond to the actual size of an object (e.g., size of a vertex in the operation Ephemeral Vertex) or to a mean value (e.g., mean size of a vertex in operation Versioned Subgraph). \( m \rightarrow \) total number of stored changes; \( |S| \rightarrow \) total size of snapshot; \( |E| \rightarrow \) eventlist size; \( h \rightarrow \) height of a tree of snapshots used in TGI. It is upper bounded by \( \log m \); \( |C| \rightarrow \) number of changes within a vertex. It is upper bounded by \( m \); \( d \rightarrow \) the degree of a vertex; \( |W| \rightarrow \) size of an ephemeral subgraph \( W \); \( |A| \rightarrow \) size of an ephemeral vertex.

edges in the case of an undirected graph) as well as a set of attributes that are attached to this particular vertex or to an edge. An ephemeral edge is similarly defined. A \( \Delta \) is a set of ephemeral vertices and edges at potentially various times instances. All kinds of graph operations can be defined on \( \Delta s \) in order to compress and make more efficient queries on time instances or time intervals. An event is the minimum change that registers a new instance of the graph. As a \( \Delta \), an event is simply the set of ephemeral vertices and edges that constitute the changes between two successive time instances. An eventlist is a sequence of successive events sorted in a chronological order. An eventlist is specified by the time interval \([t_{start}, t_{end}]\) of the respective \( \Delta s \). Finally, a snapshot at a particular time instance \( t \) is the set of all ephemeral vertices and edges at time instance \( t \).

Following the above definitions, we provide a qualitative comparison between different methods in Table 5.2. Column “Space” refers to the space required by each model while the rest of the columns refer to the access time required by each respective operation.
5.2. Storage Model

**Ephemeral** operations correspond to accessing a particular specified object (graph, subgraph or vertex) at a particular time instance while **Versioned** operations correspond to accessing a particular object in a time interval. The 1-Hop operation corresponds to accessing all adjacent vertices of a particular vertex. The OPTIMAL row corresponds to the ideal space and time access costs for the operations in the worst-case.\(^1\)

The LOG method corresponds to a single initial snapshot with an event-list that stores all changes. The COPY method corresponds to a snapshot for each change without eventlists. COPY corresponds to storing each snapshot as a separate graph in a graph database system. One could combine these methods (COPY+LOG) by allowing a sequence of snapshots with eventlists that record the changes between them. In the following paragraphs, we describe the space cost required by the TGI and \(G^*\) approaches. Furthermore, we outline the time cost for the operations in the first half of Table 5.2 (the rest follow the same logic and are defined similarly).

TGI is based on the COPY+LOG idea which, however, is considerably tuned so that it allows a hierarchical structure of snapshots combined with partitioning of eventlists into small chunks in order to achieve better locality. Furthermore, the system supports lists of different instances of diachronic vertices within the snapshots (“vertex chains”) to facilitate vertex-centric operations. In addition, one of its merits is the dynamic partitioning of the graph. We choose not to incorporate it in this comparison for reasons of uniformity since all other solutions do not support such an explicit partitioning process but consider it as an additional external mechanism. Finally, updates in TGI are only supported in batch mode and the system does not allow for online small changes.

Before outlining the space and time complexity attained by TGI, we note that the parameter \(h\) corresponds to the height of the hierarchical tree structure employed by the TGI approach. On each level of the hierarchical structure, TGI stores the delta difference between each parent node from its child, thus bringing the total space cost to \(O(hm)\). To make an edge insertion we have to materialize the hierarchy of differences which results in a cost of \(O(hm)\). Obtaining a snapshot in TGI is accomplished by traversing a root-to-leaf path in the tree structure and materializing any delta associated to the snapshot on each node. This results in a \(O(h|S| + |E|)\) cost with the \(|E|\) factor corresponding to any remaining changes in the leaf-eventlist stored in the leaf that must be applied to obtain the snapshot. The Ephemeral Vertex operation has the same cost since it is performed similarly to a snapshot retrieval with the difference being that the system only retrieves the subset of deltas associated to the queried node. Finally, in the Versioned Vertex operation, the

---

\(^1\)For \(G^*\), TGI and our solution (and to a lesser extent for the other methods), one could indeed describe the complexity w.r.t. a variety of parameters and provide a more detailed description. However, doing so would certainly not permit the direct comparison between the methods and would thus invalidate the very reason for which this table is provided.
system makes use of the vertex chains to obtain the eventlists corresponding to each change in the node in a given query interval, resulting in an $O(|C||S|)$ cost for the $|C|$ eventlists that need to be filtered for the queried node.

$G^*$ is a combination of a vertex-centric approach and the COPY+LOG method, where differences are stored in a compact way by employing indirection. The system achieves that by employing an index called Compact Graph Index (CGI). CGI manages collections of vertex version locations on disk (VL maps), with each vertex version being represented once for each combination of graphs that it exists in (see Section 5.4.1 for more practical details). This means that, for successive snapshots, only differences are stored. As a result, this fact complicates logging of operations between snapshots but to a small degree (the authors do not consider it, since their focus is only on the storage of a rather small number of successive snapshots).

$G^*$ stores each of the $m$ changes only once and employs CGI to index the different versions that occur. The number of VL maps increases quadratically with the number of graphs [76] and since $G^*$ employs logging to store all intermediate history between successive snapshots, the space cost of $G^*$ is $O(m + \frac{m^2}{|E||C|})$. The edge insertion cost of $G^*$ is $O(\lg m + |E| + \frac{m}{|E|})$ since it follows a similar approach to the COPY+LOG method with the addition of traversing the index and applying any logged updates. In order to retrieve a snapshot, $G^*$ iterates through the VL maps and retrieves all vertices that are related to the queried time instance. It then applies any logged updates before returning the snapshot resulting in a total cost of $O(|C|\frac{m}{|E|} + |E| + \lg m + |S|)$. The same procedure is followed for the Ephemeral Vertex operation but instead of returning all vertices for the queried time instance, the system limits its query to the given vertex. Finally, in the Versioned Vertex operation $G^*$ employs the same procedure but may require visiting multiple paths in the index since the query refers to a time interval instead of a particular time instance.

We provide an outline of the complexities attained by HiNode. The space cost of HiNode is linear to the number of changes (Theorem 10). The time complexity to insert an edge is the same as the one for writing an attribute (see the WriteAttribute operation in Theorem 10). First, we obtain the relevant diachronic node by traversing a B-Tree in logarithmic time and then we insert the edge. However, a more accurate description of the complexity can be attained by noting that the number of changes in the node is $|C|$ and, since $|C|$ is bounded by $m$, we obtain the final result. To obtain a snapshot, we perform multiple ReadVertex operations on all vertices that are “alive” at that particular time instance. The cost of retrieving a single vertex is $O(\log|C|)$, which refines the $O(\log_B m)$ definition in Theorem 10. Since, the total size of the snapshot is $|S|$, the total cost is $|S|\log|C|$. In order to read an ephemeral vertex we need to locate its diachronic node using the B-Tree (log $m$), retrieve the vertex at that particular time instance (log $|C|$) and read its attributes and
5.3. Query Processing

In this section, we start by discussing snapshot materialization and graph traversal, which are fundamental steps in historical queries. Then, we discuss various versions of shortest paths based on the underlying notion of time.
Finally, we refer to graph sampling as an argument in favor of efficiency in local queries.

### 5.3.1 Core Algorithms for Global Queries

We begin by outlining snapshot materialization of the graph at a specific time instance and how to execute a graph traversal algorithm (e.g. DFS/BFS) for a given source vertex and time instance.

**Theorem 11.** Given $G$ we can materialize a specific snapshot $G_t = (V_t, E_t)$ at time instance $t$ in $O(|V_t| \log_B m + S_B)$ time (I/Os) where $S$ is the total size (attributes and edges) of all vertices in $V_t$.

*Proof.* Creating a snapshot in HiNode is only a logarithmic factor away from the optimal while it compares favorably with TGI and $G^*$ (see Section 5.2.4). We begin by executing a stabbing query on $I$ to retrieve all the vertices that exist on the time instance $t$. Afterwards, we perform a `ReadVertex` operation on each of the returned diachronic nodes to obtain the final result. The initial stabbing query requires $O(\log_B m + |V_t|/B)$ time and each subsequent `ReadVertex` operation takes $O(\log_B m + |u_t|/B)$ time where $|u_t|$ is the size of the vertex $u$ at time $t$. Let $S$ be the total size of all vertices in $V_t$. Since there are $|V_t|$ `ReadVertex` operations and $S \geq |V_t|$, this brings the total time to $O(|V_t| \log_B m + S_B)$. \hfill \qed

For DFS, the straightforward way would be to materialize the snapshot and run DFS on the snapshot. However, by using a stack data structure and repeatedly using `ReadVertex` in each newly-visited node we can naturally apply DFS on the diachronic nodes without having to resort to materialization. We only have indications as to whether this is more efficient than materializing the corresponding snapshot. In particular, preliminary results, as indicated in the next section show that in principle materialization has larger costs (e.g. requires the explicit storage of the snapshot) and it is expected that HiNode will be more efficient as indicated in Section 5.4.3. Additionally, experimental results suggest that when more than one snapshots are required (a time interval and not a time instance) our method is certainly more efficient.

**Theorem 12.** Given $G$ we can perform a depth-first search on a specific snapshot $G_t = (V_t, E_t)$ at time instance $t$ starting from a source vertex $v$ in $O(|V_t| \log_B m + S_B)$ time (I/Os) where $S$ is the total size (attributes and edges) of all vertices in $V_t$.

*Proof.* To perform a depth-first search an external stack data structure [95] is required. The external stack is the external memory equivalent of an internal memory LIFO data structure and supports insertions (push) and deletions (pop) in $O(1/B)$ amortized time. As a first step, we retrieve the diachronic node of $v$ using $B$ in $O(\log_B m)$ time and push the node to the stack. We then
5.3. Query Processing

iteratively pop the node in the top of the stack, mark it as visited and perform a \texttt{ReadVertex} on the acquired node. For each outgoing edge we push the respective node in the stack and repeat the same procedure until the stack is empty. The worst case for this algorithm occurs when \(G_t\) is connected and thus all \(|V_t|\) vertices are eventually inserted and deleted from the stack.

The push and pop require \(O(|V_t|/B)\) amortized time in total, while all the \texttt{ReadVertex} operations require \(O(|V_t| \log_B m + \frac{S}{B})\) time (Theorem 11), which brings the total cost to \(O(|V_t| \log_B m + \frac{S}{B})\) time.

Finally, we move to the discussion of finding the shortest path from a single source \(v\) to a single target \(u\). We base our algorithms on the well-known Dijkstra [34] algorithm for solving this problem which stops as soon as it discovers the shortest path between these two nodes. We provide two different versions of the problem based on the underlying notion of time: a) \textit{time travel shortest paths} and b) \textit{time instance shortest paths}. In the former variant, the user provides the source node \(v\) and the target node \(u\) as well as a time interval \([t_s, t_e]\) and asks the shortest path between \(v\) and \(u\) that uses edges and nodes valid throughout the time interval \([t_s, t_e]\). This means that the shortest path is allowed to use edges and nodes that do not coexist at the same time instance (we allow for time traveling when traversing the shortest path - e.g. in the case of airline travel with connection cities, a traveler is usually not interested in all the flight paths coexisting at the same time). In the latter variant, the user provides again the same input but expects to find a shortest path between \(v\) and \(u\) that is valid at a particular time instance within the time interval \([t_s, t_e]\), that is time travel is not allowed. For the latter variant, one could construct all the \(t_e - t_s + 1\) snapshots and run the Dijkstra algorithm in each one of them reporting at the end the one with the minimum length, which is a clear waste of time when these paths have a lot in common. In the time travel shortest path, materialization does not lead to a straightforward algorithm.

\textbf{Theorem 13.} Given \(G\), \([t_s, t_e]\) and nodes \(v, u\), we can find a time travel shortest path between \(v\) and \(u\) within the time interval \([t_s, t_e]\) in \(O((|E_{s,e}| + |V_{s,e}| \log |V_{s,e}|) \log_B m)\) time (I/Os), where \(V_{s,e}\) and \(E_{s,e}\) are the set of nodes and edges respectively that are valid in the time interval \([t_s, t_e]\).

\textbf{Proof.} The algorithm is the same with that of Dijkstra with the exception that we consider edges and nodes that are valid in the time interval \([t_s, t_e]\). In particular, when the algorithm considers the neighbors of a node \(w\) to update their distance, it chooses nodes that are valid in \([t_s, t_e]\). In addition, the length of the edge between \(v\) and its neighbor \(z\) is the minimum length among all edges between \(v\) and \(z\) that are valid in the time interval \([t_s, t_e]\). Note that the distance (\texttt{dist}) and the predecessor labels (\texttt{prev}) can be stored separately maintaining pointers to the index and not the diachronic nodes. See Algorithm 5 for an extensive description. In the description of the algorithm, it
Algorithm 5 TimeTravelSSSP($G$, $v$, $u$, $[t_s, t_e]$)

Input: evolving graph sequence $G$, time interval $[t_s, t_e]$, source $v$ and target $u$
Output: The time travel shortest path $p$ from $v$ to $u$ in the range $[t_s, t_e]$

▷ dist[$w$] stores the minimum distance from $v$ to $w$
▷ prev[$w$] stores the predecessor of $w$
▷ Q is an external memory priority queue

1: Using $\mathcal{I}$ acquire the list $L$ of nodes that are valid in $[t_s, t_e]$ (simple pointers to them)
2: for each vertex $w \neq v$ in $L$ do
3: dist[$w$] ← $+\infty$
4: initialize the predecessor of $w$: prev[$w$] ← NULL
5: InsertKey(Q, $w$, $+\infty$)
6: end for
7: while target $u$ has not been reached do
8: $w$ ← ExtractMin(Q)
9: for each valid neighbor $z$ of $w$ in $[t_s, t_e]$ do
10: Let $\Lambda$ be the set of lengths of valid edges $(w, z)$
11: Let length$(w, z) = \min \Lambda$
12: newd ← dist[$w$] + length$(w, z)$
13: if newd < dist[$z$] then
14: dist[$z$] ← newd
15: prev[$z$] ← $w$
16: DecreasePriority(Q, $z$, newd)
17: end if
18: end for
19: end while
20: return dist[$u$], prev[$u$]

is the definition of the $\Lambda$ set that introduces the time travel in the computation of the shortest paths. Choosing the minimum among all lengths in $\Lambda$ returns at the end the time travel shortest path between $v$ and $u$ since any other choice would lead to a shortest path with at least the same length (this is an exchange argument for the greedy method of choosing lengths). We use standard terminology for the description of the algorithm while the shortest path can be generated by backtracking from $u$.

The time complexity is $O\left((|E_{s,e}| + |V_{s,e}| \log |V_{s,e}|) \log_B m\right)$, where the factor $\log_B m$ is derived from accessing each field in the diachronic node, while the rest of the time complexity comes from the Dijkstra algorithm implemented with an efficient priority queue [20].

As previously mentioned, materialization cannot be used in this case. Employing Algorithm 5 in the TGI would require the use of the lists of different instances of the diachronic nodes. This would mean that all accesses would be non-local and thus it is expected that the efficiency would be deteriorated. In addition, it requires more work to find all edges that are valid in
$[t_s, t_e]$ since we need to traverse a list of nodes to find them. $G^*$ is expected to be more efficient than TGI but due to the level of indirection that we need to access for finding the edges it is expected that it will be less efficient than HiNode, especially when the time interval $[t_s, t_e]$ is not small (e.g., if it contains 3 time instances).

The time instance shortest path requires more care since we need to discover a shortest path between $v$ and $u$ that is valid in one time instance without resorting to materialization, which is the first thing that comes to mind and gives rise to a natural algorithm.

**Theorem 14.** Given $G$, $[t_s, t_e]$ and nodes $v$, $u$, we can find a time instance shortest path between $v$ and $u$ within the time interval $[t_s, t_e]$ in $\tilde{O}( (|E_{s,e}| + |V_{s,e}| \log |V_{s,e}|) \log_B m )$ time (I/Os), where $V_{s,e}$ and $E_{s,e}$ are the set of nodes and edges respectively that are valid in the time interval $[t_s, t_e]$.

**Proof.** The algorithm resembles that of Dijkstra with the exception that each node is accompanied by an interval in which it is valid (always within the given interval $[t_s, t_e]$). This means that in the priority queue we do not store each node only once but as many times as the different instances of the node within the time interval $[t_s, t_e]$. A description is shown in Algorithm 6. The major difference is that we handle many copies of the same node in the priority queue that correspond to many different time intervals (but not necessarily time instances). In addition, another thing to notice is that the stopping condition is not related to the target $u$ but requires the priority queue $Q$ to be empty. This is because when a node is processed it does not mean that it is processed for its full range of its valid interval but only for a subinterval. One could also define as a stopping condition that all instances of $u$ corresponding to different time intervals are visited.

The time complexity is $\tilde{O}( (|E_{s,e}| + |V_{s,e}| \log |V_{s,e}|) \log_B m )$, where the factor $\log_B m$ is derived from accessing each field in the diachronic node, while the rest of the time complexity comes from the Dijkstra algorithm. The $\tilde{O}$ notation represents multiplicative factors based on the implementation of the hash maps. Finally, note that each interval corresponding to an object among the $|V_{s,e}| + |E_{s,e}|$ different objects creates only $O(1)$ new records in the data structures used in Algorithm 6.

### 5.3.2 Graph Sampling: A local query case-study

We use graph sampling [4, 55] as a means to show that local operations on graphs are of critical importance. Thus, there is a need to be able to access efficiently subgraphs of a graph without resorting to materialization. Graph sampling is a technique related to picking a subset of vertices and/or edges from a given graph aiming at preserving and/or estimating certain desired graph properties. In this way, the new smaller graph is similar with respect
The time instance shortest path

\begin{algorithm}
\begin{algorithmic}
\State \textbf{Input:} evolving graph sequence $\mathcal{G}$, time interval $[t_s, t_e]$, source $v$ and target $u$
\State \textbf{Output:} The time instance shortest path $p$ from $v$ to $u$ in the range $[t_s, t_e]$.
\Procedure{InstanceSSSP}{$\mathcal{G}, v, u, [t_s, t_e]$}
\State $\triangleright$ dist and prev are hash maps.
\State 1: Using $I$ acquire the list $L$ of nodes that are valid in $[t_s, t_e]$.
\For{each vertex $w \neq v$ in $L$}
\State 3: $\text{dist}[w_{[0, +\infty]}] \leftarrow +\infty$ dist is a hash map.
\State 4: $\text{prev}[w_{[0, +\infty]}] \leftarrow \text{NULL}$
\State 5: InsertKey($Q, w_{[0, +\infty]}, +\infty$)
\EndFor
\While{priority queue $Q$ is not empty}
\State 7: $w_{[t,t']} \leftarrow \text{ExtractMin}(Q)$
\For{each neighbor $z_{[t_s,t_e]}$ of $w$ such that $[t, t'] \cap [t_s, t_e] \neq \emptyset$}
\If{$t \leq t_z \leq t' \leq t'_z$}
\State 10: newd \leftarrow dist$\left[w_{[t,t']}\right] + \text{length}\left(w_{[t,t']}, z_{[t_s,t_e]}\right)$
\State 11: oldd \leftarrow dist$\left[z_{[t_s,t_e]}\right]$
\If{newd < oldd}
\State 12: Remove $z_{[t_s,t_e]}$ from dist and prev
\State 13: Add to dist, $z_{[t_s,t_e]}$ with value newd
\State 14: Add to dist, $z_{[t',t_e]}$ with value oldd
\State 15: Add to prev, $z_{[t_s,t_e]}$ with value $w_{[t,t']}$
\State 16: Add to prev, $z_{[t',t_e]}$ with value $w_{[t,t']}$
\State 17: RemoveKey($Q, z_{[t_s,t_e]}$)
\State 18: InsertKey($Q, z_{[t_s,t_e]}, \text{newd}$)
\State 19: InsertKey($Q, z_{[t_s,t_e]}, \text{oldd}$)
\EndIf
\EndIf
\State 20: Similarly as in Lines 10-23, for the cases $[t, t'] \supseteq [t_s, t_e], [t, t'] \subseteq [t_s, t_e]$ and $t_z \leq t \leq t'_z \leq t'$
\EndFor
\State 24: \Return dist$[u]$, prev$[u]$
\EndProcedure
\end{algorithmic}
\end{algorithm}

to certain properties to the full one. Thus, an algorithm may be applied to
the smaller graph to compute these properties for the full graph, leading to
improved efficiency. The main motivating example for graph sampling is the
lack of data (e.g., API rate limits in Twitter) or lack of resources (e.g., time)
to access the data (e.g., the huge graph of all followers in Twitter). Although
sampling can be tackled by optimization methods, these assume full access
to the graph in the first place which as we said earlier is either not possible
or time consuming. As a result, we focus only on simple approaches that are
tailored to our framework. In particular, we assume that the evolution of the
graph is fully stored in HiNode but due to time constraints we wish to access
only a part of the graph at a particular instance or time interval in order to
estimate certain properties. Note that [4] has studied graph sampling in a streaming framework which can only be seen orthogonally to our study.

The most important graph sampling techniques include Vertex Sampling (VS) and Traversal Based Sampling (TBS). Let \( G = (V, E) \) be a simple graph. In the VS technique, a subset \( V' \subseteq V \) is chosen randomly as well as all edges between these vertices that belong to \( E \), that is \( E' = \{(u, v) : (u, v) \in E, u, v \in V'\} \). A major version of this technique is Vertex Sampling with Neighborhood (VSN), where initially a set of vertices \( \tilde{V} \) is chosen and then \( E' \) is the set of all edges that are incident to \( \tilde{V} \). Finally, in TBS, a sampler starts with a set of initial vertices and then extends the sample by following edges from vertices already visited by employing various strategies (e.g., randomly, BFS, DFS).

We now discuss how graph sampling fits into our framework. In graph sampling one needs to access a limited number of vertices/edges bounded by a predefined budget \( L \) which is reduced each time an edge or a vertex is sampled. If the operation is applied at a single snapshot, then one can simply materialize this snapshot and then apply the sampling procedure to it. However, this is contradictory since it requires the full access of the instance which cancels all advantages of sampling. As a result, one cannot employ methods that store historical graphs that are based on materializing snapshots in order to support such operations. It is more appropriate in this case to materialize single vertices, which is the main strong point of vertex-centric storage techniques, like ours.

To clarify this point we look at the problem of computing the degree distribution when graph sampling is employed and comparing \( G^* \), TGI and HiNode on such a scenario. VS and Random Walks (RW) have a pretty good performance in approximating the degree distribution of the underlying network (directed or undirected) [101] since they are unbiased estimators and their mean squared error is rather small. Estimating the degree distribution in a specific time instance for both methods requires sampling randomly vertices at the specific snapshot and then, in the case of RWs, visiting adjacent vertices. TGI [66] would first construct the snapshot at this time instance and then move to the sampling process. On the other hand, \( G^* \) [76] and our method would only process the vertices that are sampled. This is more efficient when the budget \( L \) is small (e.g., \( \leq 10\% \) of the size of the sampled graph). When comparing \( G^* \) and our solution, \( G^* \) is expected to be slightly more efficient in the case of single instance VS or RW since decoding each vertex for a single time instance is usually simpler (although it has the intermediate level of indirection which HiNode avoids). On the other hand, our solution is expected to be more suitable in the case of intervals of time instances since for each vertex, \( G^* \) requires more I/Os than HiNode.

In a slightly different scenario, let us assume that we wish to find the degree distribution of a graph in a given time interval. For the VS method, TGI
finds each vertex by reconstructing the snapshot and then it uses the version chains that connect all instances of a vertex in a list to speed up the processing. Apart from the obvious problem with the reconstruction of the snapshot when the budget $\mathcal{L}$ is small, the vertex chains are not packed together and thus do not exhibit space locality. On the other hand, $G^*$ uses the same idea as in the case of sampling ephemeral vertices but because of the fact that all historical information of a vertex is simply packed in a block it is easy to access it. However, the high access cost remains due to the complicated indexing mechanism to access each time instance. Our solution has the simplest indexing mechanism while at the same time maintains all the history of the vertex within a single object and thus exhibiting high space locality leading to fast access times although more operations are required to decode the information within this diachronic node (note that decoding of a diachronic node is carried out in main memory).

5.3.3 Practical Considerations

The solution we proposed in Section 5.2 makes extensive use of the external interval tree data structure in order to provide efficient asymptotic bounds. We can take advantage of the fact that in practice, the size of an individual diachronic node and the count of the intervals it maintains for its attributes and edges is substantially small. In practice, we can replace $\mathcal{B}$-trees with linked lists and hash maps since we then avoid the constant factors that arise from the use of a more elaborate data structure.

More specifically, we can use a hash map to represent $\mathcal{B}$ and omit representing $\mathcal{I}$ to reduce the space overhead (to recreate a specific snapshot we simply visit all diachronic nodes through $\mathcal{B}$). Additionally, in each diachronic node $v$, we omit $\mathcal{I}_v$ and replace $\mathcal{B}_v$ with three hash map data structures (one for each set of attributes, incoming and outgoing edges respectively). Finally, we model each of the $\mathcal{A}_v$ trees corresponding to attributes with a linked list that maintains intervals. In practice, these modifications improve our runtime efficiency at the expense of not strictly following the asymptotic guarantees of Theorem 10 regarding the time complexity. Space usage is even better w.r.t. constant factors since there is no use of elaborate data structures. Note that the above modifications are predominantly suited for the case of historical graphs that do not exhibit a specific update pattern. In extreme cases of sequences that perform a very large amount of updates on a very small subset of vertices the use of linked lists may not be suitable.

Finally, our solution could be simplified considerably if only transaction time (rather than both transaction and valid time) was considered. In particular, the diachronic node would be structured as a simple list of updates in the node. The order of the changes in the list is dictated by the order of updates. This small change in the structure of the node allows for the more
efficient decoding of the diachronic node making it potentially more efficient than the decoding needed in $G^*$.  

5.4 Experimental Evaluation

In this section, we provide experimental results after incorporating HiNode in the $G^*$ parallel graph processing system. We aim to show (i) actual space savings, (ii) real execution times for local queries and (iii) real execution times for global queries, for which a vertex-centric approach may be thought to be inefficient. The experiments ran on a private cluster with 21 virtual machines (VMs). 20 VMs played the role of the $G^*$ workers, each having 1 VCPU, 5 GB RAM and 100 GB storage. One VM served as both $G^*$ master and worker having 4 VCPUs, 28 GB RAM and 500 GB storage. All the VMs were connected through a 1GBit local network. The memory is large enough so that it can hold any diachronic node in its entirety.

5.4.1 Implementation Details

In the original $G^*$ system, the indexing model is based on maintaining "(vertexID, diskLocation)" pairs for the vertices stored in each $G^*$ server. These pairs are stored in collections that are formed in an efficient way so that the overall space required by the index is reduced and is able to fully (or partially) fit in the internal memory of each server. While the index is maintained in memory, the vertices or edges and their attributes are stored on disk. A query on the $G^*$ system is converted to a structure of graph operators that are computed in a pipelining fashion. The basis of the graph operators is the vertex operator that retrieves a particular instance of a vertex along with its attributes and edges from the disk. Note that allowing for the index to reside in memory does not provide HiNode or $G^*$ with some advantage since the comparison is only between these two storage models.

We incorporate our work into the $G^*$ system by replacing the existing indexing model with the model proposed in Section 5.2 along with the practical improvements described in Section 5.3.3. The system is further modified so that it stores entire diachronic nodes on disk instead of vertices. To answer queries, the system still makes use of the vertex operator. However, the modified system retrieves an entire diachronic node from the disk and thus, it performs an operation equivalent to `ReadVertex` to obtain a particular instance of a vertex. Note that in our prototype implementation we don’t examine partitioning issues and all the experiments are run under the same partitioning policies for both $G^*$ and HiNode.

\footnote{The source code is available at https://github.com/hinodeauthors/hinode}
### Dataset Description

We use both real and synthetic datasets. The former provide insights into actual performance benefits, while the latter allow us to evaluate our approach under a wide range of configurations.

The real datasets were obtained from the Large Network Dataset Collection of SNAP [78]. The first dataset is a citation graph of the arXiv hep-th category released as a part of the 2003 KDD Cup [44]. The dataset contains citations from January 1993 to April 2003, which we use to create a sequence of monthly snapshots of the citation graph. The second dataset is similar to the first dataset as it focuses on the arXiv hep-ph category on the same time period while featuring a slightly larger count of vertices and edges. In both datasets, we omit 0.4% of the total edges due to the difficulty of mapping them to a specific snapshot (e.g., paper A cites paper B but paper B is inserted in the dataset with a later timestamp than that of A). The last dataset maintains records for all the US utility patents granted between 1963 and 1999 and their cross-citations. We build a sequence of monthly snapshots for that time period while omitting 0.04% of the edges due to insufficient date information in the dataset (e.g., withdrawn patents). In the real datasets we only focus on the edges between vertices and do not maintain any attributes (such as names or weights). A detailed overview of each real dataset is shown on Table 5.3.

The synthetic datasets follow either the Erdős-Rényi (ER) [36] model or the Barabási-Albert (BA) [8] scale-free graph model. The latter resembles real-world settings and environments more closely. To construct an ER synthetic dataset, we supply the number of vertices and edges in the first snapshot, the number of all snapshots, and the percentage of vertices and edges inserted or updated between snapshots (e.g., in a snapshot of 1000 vertices and 1000 edges an insertion rate of 5% would result in the next snapshot having 1050 vertices and 1050 edges). Vertices have a name and edges are weighted. An update is defined as the alternation of either a vertex’s name or an edge’s weight. BA sequences are created similarly. In a BA sequence, each newly inserted vertex is connected to the existing vertices, preferring those with a larger degree, with the number of edges created for each newly inserted vertex specified by a parameter.

Finally, we note that our datasets do not contain events related to the...
5.4. Experimental Evaluation

In this section, we focus on the time required for snapshot retrieval in $G^*$ and HiNode. More specifically, in both systems each worker is assigned an equal portion of each snapshot in the sequence through a round-robin vertex assignment. Table 5.4 shows the time required to retrieve either 1 snapshot, 20% of a sequence’s snapshots or 40% of a sequence’s snapshots for a single worker.

We focused on a single worker in order to cancel out the additional communication cost for reconstructing the whole snapshot, favoring in this way $G^*$ since HiNode is more space efficient.

While both solutions perform comparably in the case of a single snapshot, HiNode performs considerably better when retrieving a range of snapshots. The former case is attributed to the fact that $G^*$ is characterized by the intermediate level of indirection that we need to access for finding the edges. On the other hand, the latter case is due to HiNode requiring less I/Os than $G^*$ overall since in HiNode a vertex’s history is contained within its diachronic node, while in $G^*$ we require multiple I/Os to fetch a vertex’s history due to its indexing mechanism.

### 5.4.3 Snapshot Retrieval

Table 5.4: Snapshot(s) portion retrieval time (in seconds) for a single worker. The percentages correspond to the amount of retrieved snapshots in the sequence.

<table>
<thead>
<tr>
<th></th>
<th>1 Snapshot</th>
<th>20% Snapshots</th>
<th>40% Snapshots</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G^*$</td>
<td>7.4</td>
<td>20</td>
<td>27.5</td>
</tr>
<tr>
<td>HiNode</td>
<td>6.7</td>
<td>7.6</td>
<td>8.7</td>
</tr>
</tbody>
</table>

deletion of vertices or edges since a deletion does not essentially differ from an update. More specifically, consider an edge that is alive and that at some time instance it becomes deleted. Upon the deletion of the edge we “update” a relevant field used to mark the edge’s lifetime. Any queries regarding that edge must also take into account the value stored by that field.

### 5.4.4 Local Query Evaluation

To demonstrate the efficiency of our approach in local query evaluation, we conducted experiments that focused on the main primitive operations (or a similar variant) of VS and RW (see Section 5.3.2) and compared our approach against $G^*$. In particular, we studied the time required for two-hop neighborhood retrieval of a specific vertex, as well as, vertex sampling in general.

In the first experiment, given a source vertex and a (range of) snapshot(s) the query outputs all the two-hop neighborhoods for each instance of the source vertex. We performed two-hop neighborhood retrieval for varying

---

3Dataset - Undirected Barabási-Albert graph: Starting vertices = $1M$, edges per newly inserted vertex = 5, vertex insertions per snapshot = $2K$, snapshots = 100
HiNode improved on G* by a factor of up to 4.2 times.

In the latter experiment, we focused on retrieving a randomly chosen set of vertices from each worker. When the query was executed on a range of snapshots, all possible corresponding instances of each sampled vertex were returned. Additionally, we executed the query for different sample sizes. In a slight modification of the experiment, instead of reporting the sample itself, we aggregated the vertices reported by each worker so as to find the degree distribution of a graph in a time instance or a time interval. This modification translates to reduced communication costs since each worker only reports the cardinality of each degree count. The time required for this query can be seen in Figure 5.6. In this experiment, HiNode improved upon G* by a factor of up to 12 times, i.e., the improvement is by an order of magnitude.

Both experiments show that in the case of a single snapshot the two systems have comparable performance, whereas our approach is substantially

---

4Dataset - Undirected Barabási-Albert graph: Starting vertices = 1M, edges per newly inserted vertex = 5, vertex insertions per snapshot = 20K, snapshots = 100

5In the case of querying the 40% of the sequence for the two-hop neighborhood of the vertex with the largest degree, G* was unable to finish since it run out of memory.
Table 5.5: Space consumption in real datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Index Size</th>
<th>Data Size</th>
<th>Index Size</th>
<th>Data Size</th>
<th>Difference (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hep-Th</td>
<td>9.49</td>
<td>788.06</td>
<td>0.95</td>
<td>98.63</td>
<td>-89.99%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-87.48%</td>
</tr>
<tr>
<td>hep-Ph</td>
<td>12.33</td>
<td>859.5</td>
<td>1.17</td>
<td>102.81</td>
<td>-90.51%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-88.04%</td>
</tr>
<tr>
<td>US Patents</td>
<td>1094.41</td>
<td>23407.75</td>
<td>122.38</td>
<td>5456.63</td>
<td>-88.82%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-76.69%</td>
</tr>
</tbody>
</table>

better than $G^*$ when the query is concerned with a range of snapshots. In addition, HiNode performs increasingly better on nodes with a higher degree. These two observations stem from the fact that HiNode requires less I/Os than $G^*$, to retrieve all the instances of a vertex.

### 5.4.5 Space Consumption

The space utilization of each system for each of the real datasets can be seen in Table 5.5. The space savings are up to an order of magnitude. Our proposed solutions use approximately 90% less space for indexing and 76% to 88% less space for the data files. Recall that in our solution the index consists of a LinkedHashMap containing “(DiachNodeID, Location)” pairs.

Next we experiment with the synthetic datasets for several sequence sizes and insertion/update rates, as shown in Figure 5.7. Our main observations are as follows. (i) As previously, the space savings reach an order of magnitude. (ii) The higher the insertions or updates, the more significant the savings. This can be explained by the fact that, since all vertex and edge updates are stored in the diachronic nodes, the size of the index becomes larger only when new vertices are created in the sequence. A similar observation can be made about the data file sizes. (iii) In general, our proposed system favors sequences with a higher update-to-insertion ratio and (iv) the relative differences in space consumption remain the same across experiments with different starting vertices and edges counts. Similar observations can be drawn for BA (see Figure 5.8), where the savings in space are slightly smaller, i.e., up to 84% less space.

### 5.4.6 Time Efficiency for Global Queries

In the last part of the experiments, we measure the running time for the following global queries [76]:

**Vertex Degree Distribution (DegDistr):** for each graph snapshot, calculate the vertices with a specific vertex degree, sorted by the vertex degree in a descending order.

**Average Vertex Degree (AvgDeg):** for each snapshot, compute its average vertex degree, and give results in a descending order.
Chapter 5. HiNode: Space-Optimal Storage for Historical Graphs

| Clustering Coefficient Distribution (ClCoeff): | for each snapshot, calculate the clustering coefficient of its vertices. Report the number of vertices grouped by the clustering coefficient sorted in a descending order. Note that this is a bulk synchronous parallel (BSP) operator and more difficult to evaluate. |

| Shortest Path Distance Distribution (ShortPath): | compute the distance between a source vertex and all other vertices in a snapshot, for each snapshot. Report the count of vertices with a specific distance to the source vertex sorted in a descending order. |
5.4. Experimental Evaluation

<table>
<thead>
<tr>
<th>Sequence Setting</th>
<th>Size (MB)</th>
<th>Index Size (G*)</th>
<th>Index Size (HiNode)</th>
<th>Data Size (G*)</th>
<th>Data Size (HiNode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ins/Snap.:100K</td>
<td>Snapshots:10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ins/Snap.:10K</td>
<td>Snapshots:100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ins/Snap.:1K</td>
<td>Snapshots:1K</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5.8:** Space consumption in the BA datasets. Top: effect of granularity (starting vertices = 2M and edges per newly inserted vertex = 10). Middle: effect of updates (starting vertices = 2M, edges per newly inserted vertex = 10, insertions per snapshot = 100K, snapshots = 10). Bottom: effect of the number of snapshots (starting vertices = 10K, edges per newly inserted vertex = 1, insertions per snapshot = 10).

The results for the real datasets are reported in Table 5.6. The “DegDistr” and “AvgDeg” queries were run on all the monthly snapshots of each sequence on both systems. However, due to high memory demand by the original G* system, running the “ClCoeff” query in all snapshots of the “US Patents” dataset was infeasible. For that reason, we applied the “ClCoeff”
query in subsets of the snapshots in the “US Patents” sequence. More specifically, in Table 5.6 the symbols “∗”, “†” and “‡” represent that the query was run on the last snapshot of the sequence, the last five snapshots of the sequence and all the snapshots, respectively. The first observation that can be made is that our system is more efficient for the “DegDistr” and “AvgDeg” queries, yielding up to 30% faster response times. This can be explained by the fact that, since we retrieve entire diachronic nodes from the disk, we need to make fewer accesses on the secondary memory compared to the original system which retrieves specific instances of each vertex. These benefits outweigh the additional time overhead of reconstructing a vertex from a diachronic node in a particular time instance, thus reducing the total time cost. In the “ClCoeff” query our system has slightly inferior performance compared to the original system that can be explained by the nature of the datasets. The datasets exhibit a “cold start” phenomenon in that the first snapshots of the sequence have very few vertices and edges that in turn results to the cost of vertex reconstructions overcoming the gains of the fewer disk accesses. This is also shown in the “US Patents” dataset, where our system has better performance when the “ClCoeff” query focuses on the (quite large) five last snapshots of the sequence.

We achieve significant speedups for the synthetic datasets as well. The results for the ER sequences are shown in Figure 5.9. For all three types of queries, the maximum reduction in response time is 54%-56%.

Next, we tested the BA synthetic sequences, executing the queries on varying sequence portions. More specifically, we executed a query on the last snapshot of the sequence or on a selection of the last 5-20% of the snapshots of the sequence. Additionally, we ran the queries on non-consecutive snapshots by specifying an appropriate step size. Initially, we investigated the impact of granularity, as in the space-efficiency experiments. We ran the three queries in sequences of 10, 100 and 500 snapshots. Regarding the sequence with the 10 snapshots, since the percentages of the previous paragraph do not directly correspond to meaningful snapshot ranges, we ran the queries in the last 1, 2 and 5 snapshots. The results can be seen in Figure 5.10. In the case of querying only the last snapshot of the sequence, our method is slower since it suffers from the time overhead of reconstructing that particular snapshot. However, the time efficiency of our approach improves as
5.4. Experimental Evaluation

Figure 5.9: Time efficiency in sequences with \( \text{insertion\_rate} = \text{update\_rate} = 1\% \) (top), \( \text{insertion\_rate} = 2\%, \text{update\_rate} = 1\% \) (middle), and \( \text{insertion\_rate} = 1\%, \text{update\_rate} = 2\% \) (bottom).

The query percentage becomes higher. The effect of updates is shown in Figure 5.11. Again, for higher query percentage we achieve better performance.

Finally, we evaluated the impact of graph density on the total query time, after building sequences of varying vertex degree per newly inserted vertex. The results showed that density played no significant role as the difference...
Chapter 5. HiNode: Space-Optimal Storage for Historical Graphs

**Figure 5.10:** Effect of granularity on time. Starting vertices = 1M, edges per newly inserted vertex = 10. Insertions/snapshot = 200K/20K/4K for sequences with 10/100/500 snapshots, respectively.

**Figure 5.11:** Effect of updates on time. Starting vertices = 1M, edges per newly inserted vertex = 5, snapshots = 10.

**Figure 5.12:** Shortest path evaluation and effect of graph density. Starting vertices = 1M, vertex insertions per snapshot = 20K, snapshots = 100. The source vertex was the vertex with the highest degree.

remained relatively the same. More specifically, Figure 5.12 shows the evaluation of “ShortPath” for a sequence of 100 snapshots with varying vertex
5.5 Discussion

Multiple Universes HiNode is designed so that it can support transaction time as well as valid time. Additionally, with minor modifications, HiNode can also support multiple universes in the sense that the history has a tree-like shape. This is reminiscent of the notion of full persistence in data structures [102], in the sense that the history of the data structure is fully characterized by a tree structure. Similarly, since history is not linear, we require its explicit representation by a history tree. Instead of talking about time that implies a linear evolution we now talk about versions of graphs. New version instances are created by making updates to existent versions of the history tree. For example, let a node $v$ of the history tree corresponding to the graph of version $v$. An update at version $v$ gives a new instance that is represented by node $u$ (with version $u$) that is a child of $v$ in the version tree. We refer to the interested reader for a more detailed analysis to [102].

The crucial point is that navigation in history requires the efficient support of nearest common ancestor queries on the history tree. In this way, searching for a version $v$ in a node is equivalent to finding the version $u$ that exists in this node and is the nearest ancestor among all versions in the node of $v$.

HiNode can support a tree like history by employing an external memory data structure that can answer efficiently such nearest ancestor queries (see [23]). Having such a structure requires no other changes to our structure. $G^*$ cannot support efficiently a tree-like history even in the presence of a history tree since the structure of the diachronic nodes and the index need to change considerably. TGI cannot support such a notion of time since its snapshot architecture is rather incompatible with it.

Registering Updates An issue of our approach is the silent assumption that updates between two instances (snapshots) are readily available. Although this is trivial for a data owner (e.g. Facebook), this is not the case for users that have access to two successive instances but not to the real updates between them. Thus, the problem is that given two instances of the graph at time $t$ and $t + 1$ we must discover the differences in the nodes and the edges between these two instances in order to register them in the respective diachronic nodes.
We assume for simplicity that each node has a unique identifier that remains invariant in history. In this case, HiNode picks each node at time instance $t+1$ and finds it in instance $t$ comparing them and registering the differences in the respective diachronic nodes. For all nodes in $t+1$ that are not found in $t$ we create new diachronic nodes. The time complexity for this procedure between two time instances is $O(m)$ provided that we can search for the identifier of a node in $O(1)$ time by using hashing, where $m$ is the total number of changes stored in HiNode. Finally, in the extreme case where the identifiers are not invariant over the course of history and even in the case where they are reused we can always apply the same approach with some additional space consumption. In particular, if an existent node at time $t$ with identifier $v$ has its identifier changed at time $t+1$ to $v'$ then a new diachronic node is created that corresponds to this node. Although we use more space to save this node, since we have no other means of identifying this phenomena rather than resorting to graph isomorphism techniques, the results of any queries of the user will always be correct. Even in the case, where a node that at time $t+1$ has the identifier of another node at time $t$ will be stored correctly since all its info will be registered in the diachronic node.
Chapter 6

HiNode: A Vertex-centric Modelling Approach to Maintaining Historical Graph Data

Chapter 5 introduced the first purely entity-centric, and more specifically, vertex-centric model for maintaining graph historical data, termed as HiNode [73]. Its strongest point is that it builds upon a theoretical storage model that is asymptotically space-optimal. Its initial implementation, hereafter termed as HiNode-G∗, was based on extensions to the G∗ [76, 107] parallel graph database. This design choice incurred severe limitations regarding the efficiency and scalability of the HiNode-G∗ prototype. This chapter proposes leveraging NoSQL as the underlying database technology thus forming the second version of HiNode, called HiNode-NoSQL hereafter. However, simply switching to a different underlying technology does not necessarily imply performance improvements in retrieval tasks. The technical contribution of this work is to (i) investigate and compare different NoSQL design techniques for the HiNode-NoSQL system, and (ii) provide concrete insights into the strengths and weaknesses of each alternative in terms of supporting retrieval queries on evolving networks.

More specifically, the work in this chapter:

1. Investigates two approaches to vertex-centric modelling and storage with different strengths and weaknesses.

2. Proposes querying models on top of these models.

3. Shows that under certain cases, their relative performance can differ by several times.

4. Finally, it provides evidence that simple baseline and non-NoSQL solutions are slower by up to an order of magnitude.

The rest of the chapter is structured as follows: Section 6.1 describes the motivation behind this work. Section 6.2 describes the alternative models in detail while Section 6.3 compares the proposals through extensive experimental evaluation.
6.1 Motivation

Without loss of generality, throughout the rest of this work we will use a particular graph as a running example. In this example, each graph vertex possesses two attributes: a name and a color. Furthermore, graph vertices are connected through edges with each edge having a label and a weight attribute. Figure 6.1 depicts three consequent snapshots of the example graph.\(^1\) Snapshot \(G_1\) is obtained by changing the name of \(v_4\) in \(G_0\) from \(d\) to \(lbl\) and snapshot \(G_2\) is obtained from \(G_1\) by inserting \(v_5\) and an edge from \(v_2\) to \(v_5\).

In order to efficiently store and handle evolving graph sequences the HiNode system prototype (Chapter 5) was developed. The primary focus of the HiNode system is on storage, and more specifically how storage requirements can be minimized while efficiently supporting retrieval tasks. Retrieval tasks are either complete queries on evolving graphs or part of them, when more complex graph analyses on evolving graphs are performed.

The core idea behind HiNode’s solution is that a vertex history throughout all snapshots is combined into a set of collections called diachronic node. The diachronic node utilizes external memory Interval Trees and B-Trees to model a vertex’s history as a collection of intervals that permit geometric operations upon them. HiNode supports adding or removing vertices and attributes as fundamental operations upon which more complex operations and queries (e.g. graph traversal, shortest path evaluation etc.) are constructed. In HiNode, each change is stored \(O(1)\) times, resulting in an asymptotically optimal total space cost. As an example, for the graph in Figure 6.1 the diachronic node for \(b\) at the time of \(G_2\) would contain an Interval Tree for all changes related to \(b\) as well as three B-Trees for \(b\)’s label, color and edge to \(e\). Furthermore, due to the local handling of history, HiNode performs well on local queries and the authors further demonstrate that HiNode-G* is competitive on global queries as well [73].

---

\(^1\)For reasons of clarity edge labels, vertex colors and edge weights are not shown.
The $G^*$ parallel graph processing system [76] is, in essence, a combination of the Copy+Log storage method (see Section 5.2.4) and the vertex-centric modeling approach. The system aims to exploit the commonalities found between successive snapshots in the history of a graph to reduce the total space footprint by only storing each version of a vertex only once regardless of the number of snapshots it can be found in. The indexing mechanism of $G^*$ is a set of \text{"(vertexID, diskLocation)"} pairs for each vertex version, formed into collections based on the vertices stored by each $G^*$ server. As an example, for the graph in Figure 6.1 a $G^*$ server assigned to store vertex $b$ would store two versions of $b$: One for $\{G_0, G_1\}$ and one for $G_2$. Finally, it is worth noting that the index of $G^*$ resides in memory while the vertex data is stored on disk. HiNode-$G^*$ was essentially built on top of the $G^*$ query evaluation modules by substituting the above indexing and storage modules with a pure vertex-centric module.

This work aims to substitute the storage technology of HiNode-$G^*$ with NoSQL databases in order to avoid the underlying limitations of the $G^*$ storage engine. More specifically, the $G^*$ system was built on the premise of storing the aforementioned pairs in a distributed fashion utilizing slotted pages. HiNode-$G^*$ could only operate on datasets of restricted size due to the overall architecture and implementation of the storage and indexing modules. By adopting the use of NoSQL databases as the fundamental method of storing data the aim is to achieve a threefold goal: (i) take advantage of the natural key-value nature found in a graph’s entities (e.g. an edge has a specific name and weight at a specific time instance), (ii) take advantage of the NoSQL database’s inherent scalability especially for data-intensive applications and graphs; and (iii) to benefit from the engineering maturity of NoSQL tools. With regard to HiNode-NoSQL two design techniques (Single-Table and Multi-Table) are presented that each possesses its own tradeoffs between queries.

Finally, to underline the necessity of a NoSQL approach some indicative results are presented in Figure 6.2. Both of the two vertex-centric models for HiNode-NoSQL that will be described in Section 6.2 significantly outperform HiNode-$G^*$ even in the case of a dataset\textsuperscript{3} with a relatively limited number of vertices and edges thus solidifying the need for a more scalable approach. The adoption of a NoSQL approach in favor to HiNode-$G^*$ is not a trivial task and comes with several design and query execution issues for which their addressing constitutes the result of this work.

\textsuperscript{2}The queries executed were Average Vertex Degree, Degree Distribution and One-Hop Neighborhood Retrieval on the last snapshot of the sequence - see Section 6.3.1
\textsuperscript{3}citHep-Th SNAP Dataset [78] - see Section 6.3.1
6.2 Modelling

In this section, we propose two vertex-centric models for maintaining historical graph data and provide an implementation through Apache Cassandra\(^4\). We describe the logic behind each model and discuss advantages and potential trade-offs using the example graph sequence found in Figure 6.1.

6.2.1 Single Table Model

The first vertex-centric model, called the Single Table model (ST), uses a single table to model all history in the graph. More specifically, each row in the table corresponds to a particular vertex’s entire history and is equivalent to the contents of a diachronic node in HiNode.

```sql
CREATE TYPE histgraph.attribute (value text, start text, + end text);
CREATE TYPE histgraph.edge (label text, weight text, + otherEnd text, start text, end text);

CREATE TABLE histgraph.dianode (vid text, + start text, end text, + name list<frozen<attribute>>, + color list<frozen<attribute>>, + incoming_edges map<text, frozen<list<edge>>>),
```

---

\(^4\)http://cassandra.apache.org
In Apache Cassandra we create new data types for attributes and edges and name the single table “dianode”. The “attribute” data type corresponds to an interval of a particular value that is valid between “start” and “end”. The incoming edges of a vertex are maps that store key-value pairs of the form “(source_vertex, list_of_edges)” where “list_of_edges” is a collection of all the edges that have occurred at some time in the history between these two vertices. A similar definition applies to the outgoing edges.

Finally, we note that by this primary key declaration, the rows are partitioned to a server according to their vertex ID thus resulting in each vertex’s history completely residing in a single server. This results in each change being stored only once in the corresponding vertex row and enables faster single vertex query and local query evaluation since we avoid any unnecessary communication between servers. The downside when using ST is that whenever we need to access a vertex at a particular time instance the system must “unpack” all the collections after retrieving the relevant row. Even though this is performed on the client side it adds a significant time cost. Additionally, for application domains that are characterized by a high attribute update rate, the collection size might exceed the maximum collection size supported by Cassandra. This can be alleviated by stopping the use of a particular collection after a set amount of items and creating a new one, thus retaining the space efficiency of the ST model in general.

The contents of ST for the graph sequence of Figure 6.1 are shown in Table 6.1.

### 6.2.2 Multiple Tables Model

To avoid the time slowdowns that occur with the employment of collections, we propose the second vertex-centric model called Multiple Tables model (MT). In MT we use a single table for each vertex attribute and we differentiate between incoming and outgoing edges by using a single table for each of their corresponding attributes. Additionally, MT uses three tables to denote the “lifetime” of vertices and outgoing edges and incoming edges respectively.

```sql
CREATE TABLE histgraph.vertex (vid text,
    + start text, end text,
    + PRIMARY KEY (vid, start, end));

CREATE TABLE histgraph.vertex_name (vid text,
    + name text, timestamp text,
    + PRIMARY KEY (vid, timestamp)
    + ) WITH CLUSTERING ORDER BY (timestamp DESC);
```
CREATE TABLE histgraph.vertex_color (vid text, 
+ color text, timestamp text, 
+ PRIMARY KEY (vid, timestamp) 
+ ) WITH CLUSTERING ORDER BY (timestamp DESC);

CREATE TABLE histgraph.edge_outgoing ( 
+ start text, end text, 
+ sourceID text, + targetID text, 
+ PRIMARY KEY (sourceID, start, end, targetID));

CREATE TABLE histgraph.edge_label_outgoing ( 
+ label text, timestamp text, 
+ sourceID text, targetID text, 
+ PRIMARY KEY (sourceID, timestamp, targetID) 
+ ) WITH CLUSTERING ORDER BY (timestamp DESC, 
+ targetID DESC);

CREATE TABLE histgraph.edge_weight_outgoing ( 
+ weight text, timestamp text, 
+ sourceID text, targetID text, 
+ PRIMARY KEY (sourceID, timestamp, targetID) 
+ ) WITH CLUSTERING ORDER BY (timestamp DESC, 
+ targetID DESC);

CREATE TABLE histgraph.edge_incoming ( 
+ start text, end text, 
+ sourceID text, targetID text, 
+ PRIMARY KEY (targetID, start, end, sourceID));
6.2 Modelling

CREATE TABLE histgraph.edge_label_incoming (  
    + label text, timestamp text,  
    + sourceID text, targetID text,  
    + PRIMARY KEY (targetID, timestamp, sourceID)  
    + ) WITH CLUSTERING ORDER BY (timestamp DESC,  
    + sourceID DESC);

CREATE TABLE histgraph.edge_weight_incoming (  
    + weight text, timestamp text,  
    + sourceID text, targetID text,  
    + PRIMARY KEY (targetID, timestamp, sourceID)  
    + ) WITH CLUSTERING ORDER BY (timestamp DESC,  
    + sourceID DESC);

In MT we elect to store time instances of change on a vertex or edge attribute ("timestamp") as opposed to explicit intervals since the underlying intervals can be trivially inferred. The strong point of MT is that basic queries can be directly evaluated through querying Apache Cassandra with minimal client involvement as opposed to ST (e.g. to retrieve a particular vertex at a specific time instance the system queries “vertex_name”, “vertex_color” and “edge_label_outgoing|incoming”, “edge_weight_outgoing|incoming” for all the relevant edges). Two weak points of MT are that in order to guarantee that the system partitions all the vertex information into the same server some of the data need to be repeated. Furthermore, even though MT avoids the use of collections, it requires a greater amount of read operations to implement a basic operation compared to ST. This could result in a significant time cost if the underlying graph has vertices or edges with a large number of attributes.

The contents of MT for the example in Figure 1.2 are shown in Table 6.2.

6.2.3 Querying Modes

The two models proposed in the previous section are vertex-centric and thus inherently suited for the execution of local queries. In order to adequately support global type of queries (i.e. queries that involve a significant part of a snapshot’s vertices), the two models offer two querying modes for the retrieval of all vertices relevant to a specified query.

Let \([t_s, t_e]\) be a specified time range for which a query is about to be executed. In the first mode (termed retrieve_all), and regardless of the given time range, we retrieve all vertices from each model and then perform a client-side filtering operation, where we discard any vertices that do not belong in \([t_s, t_e]\) (Algorithm 7).

In the second mode (termed retrieve_relevant), each model first determines the vertices that are “alive” at \([t_s, t_e]\) and then retrieves them. More
specifically, in the ST model, we first query for the “start” and “end” timestamp of each diachronic node in “dianode” and then, we retrieve from “dianode” all diachronic nodes whose “start” or “end” value belong in \([t_s, t_e]\), while in the MT model we follow a similar approach using the vertex table (Algorithm 8). Mode retrieve_relevant takes into account the provided time range and is more versatile compared to retrieve_all which should be mainly used for queries that involve the majority of the vertices in the sequence (i.e. “What is the average vertex degree in each snapshot of the sequence stored so far?”).

While in ST the implementation of retrieve_relevant is straightforward, MT requires additional work since retrieving a particular (set of) attribute(s) during a certain time interval \([t_s, t_e]\) would translate to a range query and the retrieval of all data with a “timestamp” value between \(t_s\) and \(t_e\) (i.e. we are not interested in any updates that occur outside \([t_s, t_e]\)). Since Cassandra does not natively permit range queries for two or more clustering columns (i.e. “start” and “end”) for the sake of efficiency, we fetch the relevant data with a timestamp larger than \(t_s\) and then filter, all data with a timestamp less than \(t_e\) at the client side. In the following experimental evaluation section,
6.3. Experimental Evaluation

In this section, we experimentally evaluate the two proposed vertex-centric models and compare their relative efficiency. We begin by providing query definitions, dataset descriptions before moving on to the models’ comparison on a single node setting. We start with a single-node setting in order not to allow automated parallelism and elasticity features of Cassandra to be involved and therefore, draw clearer observations about each model’s performance. Furthermore, we repeat the experiments in the presence of a cluster data center and report our findings. Finally, we outline the efficiency of a state-of-the-art graph DBMS in the context of historical graph data and conclude with some overall observations.

---

Algorithm 7 retrieve_all

**Input:** Time range \([t_s, t_e]\)

**Output:** Data from vertices that exist in \([t_s, t_e]\)

1: \(\text{if model} == \text{ST} \text{ then} \)
2: \(S \leftarrow \text{"SELECT * FROM dianode"} \)
3: \(\text{for each diachronic node } D_v \in S \text{ do} \quad \triangleright \text{ Performed client-side} \)
4: \(\text{if } D_v.end \leq t_e \text{ or } t_e \leq D_v.start \text{ then} \)
5: \(S \leftarrow S \setminus \{D_v\} \)
6: \(\text{end if} \)
7: \(\text{end for} \)
8: \(\text{else if model} == \text{MT} \text{ then} \)
9: \(\text{for each table } T \text{ in MT (i.e. "vertex", "edge_outgoing" etc.) do} \)
10: \(S \leftarrow \text{"SELECT * FROM } T\text{"} \)
11: \(\text{for each row } r \in S \text{ do} \quad \triangleright \text{ Performed client-side} \)
12: \(\text{if } r.end \leq t_s \text{ or } t_e \leq r.start \text{ then} \)
13: \(S \leftarrow S \setminus \{r\} \)
14: \(\text{end if} \)
15: \(\text{end for} \)
16: \(\text{Join remaining results from each table } T \text{ on vertex ID} \quad \triangleright \text{ Performed client-side} \)
17: \(\text{end for} \)
18: \(\text{return } S \)

we primarily employed the retrieve_relevant method due to its overall time efficiency (see Section 6.3.6).

---

\(^5\text{Source code available at https://github.com/akosmato/HinodeNoSQL}\)
Algorithm 8 retrieve_relevant

Input: Time range \([t_s, t_e]\)
Output: Data from vertices that exist in \([t_s, t_e]\)

1: if model == ST then
2: \(V \leftarrow \text{"SELECT \, vid, \, start, \, end \, FROM \, dianode"} \)
3: for each row \(r \in V\) do \(\triangleright\) Performed client-side
4: if \(r.end \leq t_s \, \text{or} \, t_e \leq r.start\) then
5: \(V \leftarrow V \setminus \{r\}\)
6: end if
7: end for
8: \(S \leftarrow \text{"SELECT * FROM dianode WHERE vid IN \, V"} \)
9: else if model == MT then
10: \(V \leftarrow \text{"SELECT * FROM vertex"} \)
11: for each vertex \(v \in V\) do \(\triangleright\) Performed client-side
12: if \(v.end \leq t_s \, \text{or} \, t_e \leq v.start\) then
13: \(V \leftarrow V \setminus \{v\}\)
14: end if
15: end for
16: \(S \leftarrow \text{"SELECT * FROM MT.{all tables} WHERE \{vid, sourceID, targetID\} \, IN \, V"} \)
17: Filter from \(S\) all data with timestamp \(\notin [t_s, t_e]\) \(\triangleright\) Performed client-side
18: end if
19: return \(S\)

6.3.1 Query Definitions and Dataset Description

We demonstrate the different trade-offs by each of our proposed models by performing experiments using four different queries. More specifically, we implemented the queries shown on Table 6.3 and executed them on query ranges of varying size (i.e. 1 snapshot of the sequence and 10%, 20%, 50%, or 100% of the sequence’s snapshots).

We performed experiments on real world datasets originating from the SNAP Dataset Collection [78]. We selected three datasets with the first two constituting a graph representation of an arXiv citation network for two different scientific domains; “hep-th” (from January 1990 to April 2003) and “hep-ph” (from February 1992 to February 2003) and the last one representing a cross-citation network of US utility patents granted between 1963 and 1999 (“USPatents”). In each case, we split the citations based on their date of occurrence and ended up with a sequence of monthly snapshots.\(^6\)

In addition to the aforementioned settings, we aimed to discover how the two models handle synthetic scale-free graphs. To this end, we produced a synthetic dataset that follows the Barabási-Albert (BA) [8] scale-free graph

\(^6\)Due to difficulties in assigning some specific edges to a particular snapshot we removed 0.4% of the total edges in the “hep-th” and “hep-ph” datasets and 0.04% edges of the “US-Patents” dataset.
### 6.3. Experimental Evaluation

<table>
<thead>
<tr>
<th>Query</th>
<th>Query Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex History Retrieval</td>
<td>For a given vertex ( vID ) and a given range of snapshots, return the state of ( vID ) at each snapshot</td>
</tr>
<tr>
<td>One Hop Neighborhood</td>
<td>For a given vertex ( vID ) and a given range of snapshots, return the neighbors of ( vID ) at each snapshot</td>
</tr>
<tr>
<td>Average Vertex Degree</td>
<td>For each snapshot in the query range, compute the average degree of its vertices and report it</td>
</tr>
<tr>
<td>Vertex Degree Distribution</td>
<td>For each snapshot in the query range, count the vertices with each possible degree and report them</td>
</tr>
</tbody>
</table>

**Table 6.3: Query Definitions**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Vertices</th>
<th>Edges</th>
<th>Snapshots</th>
</tr>
</thead>
<tbody>
<tr>
<td>hep-Th</td>
<td>27770</td>
<td>352807</td>
<td>156</td>
</tr>
<tr>
<td>hep-Ph</td>
<td>34546</td>
<td>421578</td>
<td>132</td>
</tr>
<tr>
<td>US Patents</td>
<td>3774768</td>
<td>16518948</td>
<td>444</td>
</tr>
<tr>
<td>Synthetic</td>
<td>5100000</td>
<td>30600000</td>
<td>100</td>
</tr>
</tbody>
</table>

**Table 6.4: Evaluation datasets.** The number of vertices and edges refer to the last snapshot of the sequence

The two models were implemented using Apache Cassandra and all experiments were performed through a client application written in Java. All the single-node experiments were run on an Intel Xeon CPU E5-2620, 64GB RAM machine while the multi-node experiments were run on two machines AMD FX-9370 and Intel Core i7-3770K with 32GB RAM each respectively. In the single-node setting the client application was run on a i7-3770K, 32GB RAM machine while in the multi-node setting in a E5-2620, 64GB RAM machine. In all cases the Apache Cassandra nodes were connected through a private 1Gbit network.

As a final note, this work serves as a study of different implementation approaches to vertex-centric models and a comparison between their behavior under different type of queries. As a result, matters of efficient query
execution (such as data distribution and replication, asynchronous query execution, paging etc.) will be tackled in future work. Furthermore, in order to maintain a design that is independent of the graph domain, we omitted using specific data types for any vertex or edge attributes.

### 6.3.2 A Baseline Approach

We implemented a modified version of the ST model that will serve as a simple baseline algorithm called HiNode-Baseline. This version, which can be conceived as a model similar to the Copy+Log model, splits the data to be indexed according to a set amount of time instances (e.g. every 20 snapshots).

For each resulting subset of the sequence’s snapshots we build an independent ST model index based only on the entities (and their data) that exist on the corresponding snapshot range. Each vertex or edge that spans multiple snapshot ranges is indexed in each ST model for which it is valid. The resulting HiNode-Baseline index is comprised of multiple smaller ST indices each corresponding to its own set of assigned snapshots that are akin to smaller subsequences. In the event of a query that spans multiple snapshot ranges, we query each ST index for their respective answer and combine the results. The baseline approach permits a lateral comparison between our two proposed models and a model designed on vertex-centric principles that also incorporates time as a design mechanism.

### 6.3.3 Results in a Single-Node Setting

We begin by reporting the results for the two local queries of Table 6.3 (VerHist and OneHop) in Figure 6.3, where the range of queried snapshots varies from a single snapshot to the complete history. To avoid any randomness induced by the query range selection, we repeated the experiments on query ranges from approximately the start, the middle and the end of the sequence and averaged over their times. Our primary focus is to investigate the relative difference in execution time (each execution time decrease is defined with respect to the slower model in each case). The key observations drawn from Figure 6.3 are the following:

1. There is no clear winner between the two models for the two queries. ST outperforms MT in VerHist by up to 56%, 44%, 50% and 69% lower execution time in “hep-th”, “hep-ph”, “USPatents” and “Synthetic”, respectively.

2. MT executes OneHop faster than ST by up to 71%, 75%, 58% and 69% in “hep-th”, “hep-ph”, “USPatents” and “Synthetic”, respectively.

---

As an example, a sequence of 100 snapshots that gets indexed every 20 snapshots would be comprised of five smaller ST indices whereas a vertex that exists in the first 75 snapshots would only be present in the first 4 smaller ST indices.
6.3. Experimental Evaluation

**Figure 6.3:** Local queries on the datasets of Table 6.4 - Left Col.: VerHist, Right Col.: OneHop
3. ST gradually outperforms the baseline in VerHist as query ranges grow larger by up to 70%, 65%, 64% and 66% lower execution time in “hep-th”, “hep-ph”, “USPatents” and “Synthetic” respectively. In OneHop, a similar pattern occurs with ST outperforming the baseline by up to 73%, 76%, 67% and 66% lower execution time in the four previous datasets.

4. ST and MT execution times remain relatively unaffected by the query range.

The above results can be attributed to the following reasons. With regard to VerHist, ST fetches the relevant row’s contents directly from its single table “dianode”, while MT queries each of its tables for any rows related to the queried vertex thus inducing an additional slowdown due to the extra Cassandra queries compared to ST. In OneHop, we only fetch the relevant column from ST (“dianode.outgoing_edges”) and we only query the relevant table (“edge_outgoing”) from MT, thus ST is burdened by the extra cost of handling and flattening a collection. Since we follow the retrieve_relevant querying method (Lines 8, 16, 17 in Algorithm 8) and the total execution time is dominated by the time required to fetch the relevant data from Cassandra, the local query times for ST and MT remain practically the same regardless of the query range. Finally, the baseline’s time cost can be attributed to the total number of smaller indices that have to be queried as query ranges become progressively larger and span multiple snapshot ranges.

We move on to the results for the two global queries of Table 6.3 (AvgDeg and DegDistr) in Figure 6.4. The key observations to be made are:

1. MT is the best performing model in both global queries AvgDeg and DegDistr. MT outperforms ST in AvgDeg by up to 41%, 52%, 39% and 54% lower execution time in “hep-th”, “hep-ph”, “USPatents” and “Synthetic”, respectively.

2. Furthermore, MT executes DegDistr faster than ST by up to 63%, 57%, 58% and 59% lower execution time in “hep-th”, “hep-ph”, “USPatents” and “Synthetic”, respectively.

3. Baseline becomes gradually slower as the query range becomes larger.

The results can be explained by the following reasons. In both AvgDeg and DegDistr, we follow the retrieve_relevant querying method and retrieve the outgoing edges of all relevant vertices (“dianode.outgoing_edges” in ST and “edge_outgoing” in MT). Since we restrict the fetched data to these particular columns and ST is burdened by the use and flattening of collections, MT outperforms ST in each case with both models requiring more time for larger snapshot ranges due to the fact that more results are returned as part of each Cassandra query. Additionally, similarly to the local queries, the baseline approach becomes slower as the query ranges grow larger due to the
6.3. Experimental Evaluation

Figure 6.4: Global queries on the datasets of Table 6.4 - Left Col.: AvgDeg, Right Col.: DegDistr
need of fetching and handling more data (vertex “versions”) as more snapshots become involved in each query.

Overall MT can improve on ST in all queries apart from the vertex history retrieval by a factor up to 4X. In addition, the improvements over the baseline approach is of an order of magnitude, which provides evidence that the problem of efficiently designing vertex-centric storage models for NoSQL is not trivial and different design choices have a high impact on performance.

### 6.3.4 Results in a Multi-Node Setting

In this section, we study the relative performance of the two proposed models when employed in a Cassandra multi-node environment. More specifically, we measure the performance ratio of ST’s execution time compared to MT’s execution time\(^8\) for varying query ranges. In a multi-node Cassandra approach each machine serves as an independent Cassandra node that is interconnected to other similar Cassandra nodes through a peer-to-peer architecture. To avoid any optimization issues that may arise from replication strategies as well as query node destination, each Cassandra node maintains a full replica of each dataset. The results for the local queries are presented in Figure 6.5 while those of the global queries are depicted in Figure 6.6.

The main observations that can be derived are the following:

1. In the “Synthetic” dataset, each snapshot has relatively few changes with its predecessor/successor snapshot compared to the other three datasets which leads to less significant changes of the performance ratio over different query ranges.

2. In almost all cases of “VerHist” the two environment settings exhibit similar relative performance across all query ranges (with ST requiring approximately half the time of MT in most cases).

3. In the case of “OneHop” in the multi-node environment, MT becomes more efficient than ST as query ranges grow larger (by up to 3.5 times in the “hep-th” dataset).

4. Regarding “AvgDeg” in both single and multi-node environments, the relative difference of the two models tends to become smaller as query ranges grow larger (with the exception of the “Synthetic” dataset).

5. In the two larger datasets, “USPatents” and “Synthetic” the difference between MT and ST in the multi-node environment for “DegDistr” becomes more significant in the 50% and 100% query ranges (up to 2.9 times in the “USPatents” dataset).

\(^8\)e.g. a performance ratio of 2 corresponds to MT requiring half the execution time of ST
Figure 6.5: Performance ratio on local queries (Table 6.4) - Left Col.: VerHist, Right Col.: OneHop
FIGURE 6.6: Performance ratio on global queries (Table 6.4) - Left Col.: AvgDeg, Right Col.: DegDistr
### 6.3. Experimental Evaluation

<table>
<thead>
<tr>
<th>Model \ Query</th>
<th>AvgDeg</th>
<th>VerDistr</th>
<th>OneHop</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT Model</td>
<td>7.5 seconds</td>
<td>7.5 seconds</td>
<td>~0.001 seconds</td>
</tr>
<tr>
<td>Neo4j</td>
<td>9 seconds</td>
<td>8.7 seconds</td>
<td>0.1 seconds</td>
</tr>
</tbody>
</table>

**Table 6.5: Neo4j Time Cost Comparison**

6. Overall, MT still dominates in all queries apart from the vertex history, while improvements over the baseline are at the order of magnitude.

#### 6.3.5 Results Compared to a State-of-the-Art Graph DBMS

A question may arise as to whether Cassandra is the most applicable NoSQL system. To investigate this issue, we employ Neo4j\(^9\), a system which is tailored to graph data management. More specifically, we modeled the SNAP “hep-th” dataset (Section 6.3.1) using Neo4j in two ways.

First, we explicitly store each of the sequence snapshots as new, single graphs, implicitly ignoring the commonalities between vertices and edges. This resulted in a prohibitive space cost since just the two last snapshots acquired space that was nearly equal to that of the ST model for the whole history (see Section 6.3.6 - Table 6.6).

Second, we implemented a vertex-centric representation of the sequence with each Neo4j graph node maintaining all of its relevant data by utilizing range intervals in a similar fashion to the ST model and the diachronic nodes of HiNode-G\(^*\). This resulted in a comparable (albeit higher) space cost to that of ST and MT (Section 6.3.6 - Table 6.6): 61MBs which is twice as much space as the ST model and 33% more compared to MT. The main drawback however was slower query execution time for the queries and settings depicted in Figure 6.2; Table 6.5 summarizes the results.

Overall, the vertex-centric model we advocate is more efficiently implemented in state-of-the-art key-value-based NoSQL systems, such as Cassandra rather than using graph-oriented systems, which, albeit, are incapable of handling the graph evolution.

#### 6.3.6 Concluding Remarks

Complementary to the previous experiments, we investigated the potential trade-offs between the two retrieval methods defined in Section 6.2.3 (i.e. `retrieve_all` and `retrieve_relevant`). Figure 6.7 shows a selection of indicative results in the MT and ST models for two of the datasets used in this section. In both datasets and models, `retrieve_relevant` was faster than `retrieve_all` for almost all query ranges with the relative performance of the two approaches becoming smaller as the query range got wider. The two models exhibit

---

\(^9\)https://neo4j.com/
roughly similar times when handling query ranges of 100% sequence snapshots, with \textit{retrieve\_all} being slightly faster than \textit{retrieve\_relevant} in the MT model.

Furthermore, in the previous sections we focused on the relative time efficiency of the proposed models without taking into account the total space required by each of them. Table 6.6 showcases the space required by each model for each of the datasets in Table 6.4. In all cases the ST model (which more closely represents the original HiNode vertex-centric model of [73]) achieves better space utilization than its counterparts.

Our overall findings can be summarized as follows:

- The design approach to a vertex-centric system is not straight-forward. First off, ST outperforms MT in VerHist by up to 69% lower execution time with respect to the execution time of MT.

\footnote{Measured through the “nodetool” utility}
Furthermore, ST has a lower space utilization than MT by up to a 59% decrease of MT’s space.

On the other hand, MT achieves faster execution time in OneHop, Avg-Deg and DegDistr by up to a 75%, 54% and 63% lower execution times with respect to the execution time of ST (i.e., 4X, 2.17X, 2.7X speed-up), respectively.

In all cases the baseline algorithm was slower than either one or both of our proposed models by up to an order of magnitude.

Typical state-of-the-art graph DBMS are not inherently suited for handling historical graph data. In this work, we provide concrete evidence that Cassandra outperforms Neo4j.

The querying method retrieve_all displays comparable and sometimes better execution time than retrieve_relevant for 100% query range sizes but is slower for all other query ranges.

The two proposed models tend to have similar performance for local queries when operating on a single and on a multi-node environment, while their relative difference for global queries tends to become smaller as query ranges grow larger.
Chapter 7

Conclusions and Future Work

Over the past two decades there has been an ever-increasing amount of data being generated by a potentially heterogenous variety of sources such as social networks, Internet of Things (IoT), sensor networks, customer transaction logs of multinational corporations and others. Content provider services such as Youtube along with social networks such as Facebook, Twitter and others, report data generation rates by its users that far exceed conventional approaches to data storage and processing. Other disciplines such as genomics and astronomy also report a highly increasing data growth pattern that entail specialized solutions to the highly dynamic massive datasets that are produced.

This thesis studied two issues under the above overall problem setting: Efficient evaluation of certain preference queries as well as effective historical information management. More specifically, Chapter 3 dealt with the evaluation of semi-dynamic and fully-dynamic top-$k$ dominating queries and provided the first algorithms and data structures for 2-dimensional data that offered non-trivial asymptotical performance guarantees. The algorithms studied in that chapter constitute the first attempt to process top-$k$ dominating queries offering asymptotic performance guarantees for both their time and space cost. Existing work in the area is based completely on heuristic solutions built on top of access methods that work well in practice (e.g., R-trees [49]).

Since object ranking in databases is a fundamental operation with many applications, there is a multitude of interesting research directions for future work in the area:

- An interesting and challenging problem is the improvement of the fully dynamic algorithms’ update cost, by avoiding the global rebuilding technique [94].

- A second direction is to provide efficient top-$k$ dominating query processing for any number of dimensions.

- A third direction is to design efficient algorithms for the external memory model [2]. A baseline approach could be based on the successive
computation of the $k$ first layers of minima using iterative skyline computation. However, the goal is to offer more efficient algorithms with better performance bounds.

- Finally, it is worth investigating top-$k$ dominating queries under the streaming model of computation, by offering approximate results as well as accuracy vs performance trade-offs.

Chapter 4 studied the problem of dynamic 3-sided skyline queries and proposed the ML-tree which offers loglogarithmic time (per point) solutions for when the dataset input coordinates follow certain distribution classes. The improvement of the update performance of ML-tree constitutes a challenging open problem along with its adaptation to other models such as the external and cache-oblivious ones.

In the second part of the thesis that revolved around the area of historical information management, the asymptotically space-optimal prototype system HiNode was presented over two main implementations. In Chapter 5 the theoretical overview of HiNode was first suggested. HiNode used the concept of diachronic node to store any historical changes that are relevant to a vertex, in the vertex itself. Additionally, the first implementation, that was based on the $G^*$ parallel graph database, was described.

In Chapter 6 a NoSQL approach to implementing HiNode was proposed. The HiNode-NoSQL version showcased the need for specialized scalable solutions and demonstrated that there isn’t a single universally best approach to the vertex-centric modelling employed by the system. An interesting future work direction would be optimizing the query execution procedures by employing intuitive data partitioning and replication strategies and making use of asynchronous calls to the underlying NoSQL DBMS at specific points during each query evaluation. Another fruitful area for further work would be the study of a user-parameterized hybrid design model, such as one that does not denormalize vertex and edge attributes, that could lead to improved results regarding query times and space utilization.

The next two sections present future work directions with regard to the fields of preference queries and historical information management.

### 7.1 Dynamic Layers of Minima

The algorithms presented in Chapter 3 for the maintenance of the top-$k$ dominating points in a dataset with point insertions and deletions make use of the fact that at any time point, the top-$k$ dominating points reside in the first $k$ layers of minima (Lemma 2). In the fully dynamic setting the maintenance of the first $k$ layers of minima can be accomplished through the use of the global rebuilding technique.
The divide and conquer approach by Overmars and van Leeuwen [93] for the maintenance of the skyline (first layer of minima) points in a dataset, can be adapted in a non-trivial manner so that the maintenance of the first $k$ layers of minima under point updates can be achieved. More specifically, let $P$ be a leaf-oriented height balanced tree built on the points of the dataset, with each inner node containing the $k$ first layers of minima of the points stored in its subtree. Whenever an insertion or deletion of a point takes place, a traversal through a path in $P$ occurs that updates all of the $k$ layers in each affected inner node in a bottom-up fashion. At the end, the root of the tree contains the $k$ first layers of maxima for the entire dataset. This approach does not make use of the global rebuilding technique and could thus lead to improved time bounds.

### 7.2 Complete Historical Graph System

The second part of the thesis studied issues related to efficient historical information management and proposed an initial prototype system that offered asymptotical space-optimality and a variety of primitive operations through which a host of basic and complex queries could be supported.

An interesting future work direction would be the implementation of an elaborate, modular and extensively scalable graph historical information management system based on the principles of HiNode that takes advantages of the key-value nature of graph data to bring forth the underlying NoSQL solution’s advantages (e.g. fast write operations in Cassandra or MongoDB\(^1\) Join operations). Another challenge regarding the inner workings of the system would be developing efficient solutions for partitioning and deliberately duplicating information across distributed nodes in order to efficiently support elementary retrieval operations on arbitrary time instances in the past and present.

There is also a variety of objectives for the system’s query processing component that will permit efficient query execution. Having an enriched set of basic operators that can be used as building blocks for complex analytical queries, there is a need for developing optimization techniques regarding the manner the operators are combined in execution plans. Additionally, developing optimization techniques for load balancing and bottleneck avoidance would help in addressing two of the most commonly arising problems in massively parallel settings. Finally, having addressed all of the above challenges, a set of advanced analytic queries based on the developed basic operators could be then incorporated into the system thus increasing its overall functionality.

---

\(^1\)https://www.mongodb.com/
Appendix A

Scientific Work Conducted

Accepted Papers

Peer Reviewed Journals


International Peer Reviewed Conferences


Papers Under Review

Peer Reviewed Journals

2. Evangelos Kipouridis, Andreas Kosmatopoulos, Apostolos N. Papadopoulos, Kostas Tsichlas: Dynamic Layers of Maxima with Applications on Dominating Queries. Submitted to *Computational Geometry: Theory and Applications*

Bibliography


[59] Ravi Janardan. “On the dynamic maintenance of maximal points in...
[60] Christos Kalyvas and Theodoros Tzouramanis. “A Survey of Skyline...
[61] Sanjiv Kapoor. “Dynamic maintenance of maxima of 2-d point sets”.
[62] Alexis Kaporis, Christos Makris, Spyros Sioutas, Athanasios Tsaka-
[63] Alexis Kaporis, Christos Makris, Spyros Sioutas, Athanasios Tsaka-
[64] Alexis Kaporis, Apostolos N Papadopoulos, Spyros Sioutas, Kon-
[65] Udayan Khurana and Amol Deshpande. “Efficient snapshot retrieval...
[66] Udayan Khurana and Amol Deshpande. “Storing and Analyzing His-
[67] Udayan Khurana and Amol Deshpande. “Storing and Analyzing His-
[69] David G. Kirkpatrick and Raimund Seidel. “Output-size sensitive al-
[70] Donald E. Knuth. “Deletions That Preserve Randomness”. In: IEEE
[71] Georgia Koloniari, Dimitris Souravlias, and Evaggelia Pitoura. “On
[72] Maria Kontaki, Apostolos N. Papadopoulos, and Yannis Manolopoulos.
[73] Andreas Kosmatopoulos, kostas Tischlas, Anastasios Gounaris, Spy-

Bibliography


Bibliography


[116] Yajun Yang, Jeffrey Xu Yu, Hong Gao, Jian Pei, and Jianzhong Li. “Mining most frequently changing component in evolving graphs”. In: World Wide Web 17.3 (2014), pp. 351–376.


