Année: 2020

Thèse n°: 181/ST2I

Ecole Nationale Supérieure d'Informatique et d'Analyse des Systèmes
Centre d'Etudes Doctorales en Sciences des Technologies de l'Information et de l'Ingénieur

THÈSE DE DOCTORAT

SCALABLE AND ACCURATE HIGH-DIMENSIONAL SIMILARITY SEARCH: FROM DATA SERIES TO DEEP NETWORK EMBEDDINGS

Présentée par
Karima ECHIHABI

Le 28/07/2020

Formation doctorale: Informatique
Structure de recherche: IRDA, Rabat IT Center

JURY

Professeur Bouchaib Bounabat PES, ENSIAS, Université Mohammed V, Rabat	Président
Professeure Houda Benbrahim PH, ENSIAS, Université Mohammed V, Rabat	Directrice de thèse
Professeur Themis Palpanas Professeur, Université de Paris, France	Co-Encadrant de thèse
Professeur Mohammed Ramdani PES, FST, Université Hassan II, Mohammedia	Rapporteur
Professeur Mohamed Lazaar PH, ENSIAS, Université Mohammed V, Rabat	Rapporteur
Professeure Zohra Bakkoury PES, EMI, Université Mohammed V, Rabat	Rapportrice
Professeur Mostapha Zbakh PES, ENSIAS, Université Mohammed V, Rabat	Examinateur

Abstract

The world is drowning in a big data tsunami of high-dimensional objects that need to be analyzed in order to identify useful patterns and extract new knowledge in domains as varied as agriculture, medicine, cybersecurity, seismology, astrophysics, manufacturing, finance, and others. In response to these needs, it is imperative to build analytical systems that truly support interactive exploration on datasets containing terabytes of high-dimensional objects, with dimensions reaching hundreds to thousands.

A fundamental and challenging operation called similarity search is the main bottleneck of many critical data processing tasks such as data cleaning, data integration and big data analytics (e.g., outlier detection, frequent pattern mining, clustering, and classification). A number of exact and approximate approaches have been proposed in the literature to support similarity search over massive data series collections.

In this thesis, we unify and formally define the terminology used for the different flavors of the similarity search problem. We present a similarity search taxonomy that classifies methods based on the quality guarantees they provide for the search results, and that unifies the varied nomenclature used in the literature. Following this taxonomy, we include a survey of similarity search approaches supporting exact and approximate search, bringing together works from the data series and multidimensional data research communities. We propose extensions to existing data series indexes that can answer approximate queries with guarantees and that outperform popular state-of-the-art techniques such as LSH, kNN graphs and quantization-based inverted indexes in many scenarios. We also design and conduct the two most exhaustive experimental evaluations in the field covering both exact and approximate techniques. Building upon the deep insights gained from both studies, we propose Hercules, a new algorithm that outperforms the state-of-the-art similarity search approaches in-memory and on-disk.

Our work has far-reaching fundamental and practical implications. We demonstrate that it is possible to design efficient high-dimensional vector similarity search algorithms

with theoretical guarantees on the quality of the answers, and we thus offer a more promising alternative to the two current trends in the literature: (i) LSH-based algorithms that support guarantees, but are relatively slow, and (ii) kNN graphs and inverted indexes, which are relatively fast, but do not provide theoretical guarantees. This finding paves the way for very exciting new developments which will lead to efficient solutions that can support critical analytical tasks such as brain seizure detection, cyber-attack prevention, transportation management and data cleaning automation.

Acknowledgements

My path to graduate school has been rather unconventional. After working in industry for over fifteen years as a software engineer and consultant, I decided four years ago to pursue a PhD degree. Although I have always kept a foot in the academic door teaching courses in my field of expertise, a recent experience as a professor at Mundiapolis University helped me rekindle my passion for an academic career.

I think pursuing a PhD degree was one of the toughest and best decisions I have ever taken in my life. I expected the journey to be challenging but did not realize how satisfying it would also prove to be. I feel grateful to have been able to take the time to dive deep into a topic that fascinates me. I also feel honored to have advanced the state-of-the-art in my field.

I would not have been able to bring this project to fruition without the wise guidance of my supervisors and the unconditional support of my family, friends and colleagues.

I am particularly indebted to Prof. Themis Palpanas from the University of Paris who has taught me how to conduct world-class research despite the challenges involved with remote supervision. He inspired me to aim high, helped me gain access to the resources I needed to succeed and encouraged me during difficult times. I also deeply thank Prof. Houda Benbrahim from ENSIAS, Mohammed V University who has been very supportive of the idea of a joint-supervision and for helping me navigate the world of academic research. I also extend my thanks to my collaborators Dr. Kostas Zoumpatianos from Harvard University and Professors Panagiota Fatourou From the University of Crete, Anastasia Bezerianos and Theophanis Tsandilas from the University of Paris-Sud and Salima Benbernou and Mourad Ouziri from the University of Paris for generously sharing their time and expertise. Last but not least, I would like to thank my colleagues from the diNo lab at the University of Paris for making my visits to Paris truly memorable. I cite in alphabetical order Paul Boniol, Dr. Laura Di Rocco, Dr. Anna Gogolou, Dr.

Michele Linardi, Wissam Maamar-Kouadri, Botao Peng, Frederico Roncallo and Sabiha Tahrat.

I am forever grateful to my parents Zainab and Lahoussaine for their unconditional love and for instilling in me, at a young age, strong ethics and a passion for learning, my loving husband Ahmed for his unwavering support, and my son Ismail for always cheering me up and helping me maintain a balanced life despite the great demands of graduate school. I also wish to express my gratitude to my siblings Sofia, Abdelhamid and Ali for their love and advice throughout the years. Special thanks also go to my in-laws Zahra, Mohammed, Fatima, Hajar, Malak, Malika, and Rachida, my nieces and nephews Maryam-Aya, Sarah-Nour, Youssef, Fatima-Zahra, Mohammed and Khadija for the sunshine their bring to my life.

Above all, I owe everything I have ever achieved to Allah Almighty. May Allah accept this deed and help me seek and spread beneficial knowledge.

Contents

1 Introduction **5**

1.1 Problem Overview 6

1.2 Main Contributions 10

1.3 Outline 14

2 Preliminaries **16**

2.1 Definitions 17

2.1.1 Data Types and Distance Measures 17

2.1.2 Similarity Search Queries 21

2.1.3 Similarity Search Methods 23

2.2 A Taxonomy 25

2.3 Conclusion 27

3 Exact Similarity Search **28**

3.1 Main Contributions 29

3.2 Approaches 30

3.2.1 Summarization Techniques 31

3.2.2 Similarity Search Methods 33

3.2.3 Examples 38

3.3 Experimental Evaluation 40

3.3.1 Environment 40

3.3.2 Experimental Setup 40
3.4 Results 46
3.4.1 Parametrization 46
3.4.2 Evaluation of Individual Methods 47
3.4.3 Comparison of the Best Methods 51
3.5 Discussion 67
3.6 Conclusions 71

4 Approximate Similarity Search 72
4.1 Main Contributions 73
4.2 Approaches 74
4.2.1 Summarization Techniques 76
4.2.2 Approximate Similarity Search Methods 77
4.3 A New Class of Approximate Search Techniques 81
4.4 Experimental Evaluation 84
4.4.1 Environment. 84
4.4.2 Experimental Setup 84
4.5 Results 87
4.5.1 Parametrization 87
4.5.2 Indexing Efficiency 88
4.5.3 Query Answering Efficiency and Accuracy: in-Memory Datasets . 89
4.5.4 Query Answering Efficiency and Accuracy: on-Disk Datasets 92
4.6 Discussion 104
4.7 Conclusions 108

5 Hercules: A New Similarity Search Technique 110
5.1 Main Contributions 111

5.2 Related Work 112
5.2.1 SAX 112
5.2.2 DSTree 113
5.3 Hercules 116
5.3.1 Indexing with Hercules 116
5.3.2 Query Answering with Hercules 121
5.3.3 Complexity Analysis 130
5.3.4 Proofs 131
5.4 Experimental Evaluation 134
5.4.1 Environment 134
5.4.2 Experimental Framework 135
5.5 Results 138
5.5.1 Parametrization 138
5.5.2 Exact Query Answering 139
5.5.3 Approximate Query Answering 142
5.6 Conclusions 148

6 Conclusions and Future Work **150**

6.1 Key Contributions 151
6.2 Future Work 152

Bibliography **156**

List of Figures

1.1 Similarity Search over Collections of High-Dimensional Objects 8

1.2 Data Series Similarity Search as a k-NN Query 9

1.3 A simplified architecture for a neural network with a 4-dimensional embedding layer 10

2.1 Different flavors of the similarity search problem 25

2.2 Taxonomy of similarity search methods (• indicates our extensions to existing techniques) 26

3.1 Summarizations 32

3.2 Answering a similarity search query using different access paths 38

3.3 Leaf size parametrization 44

3.4 Buffer size parametrization 45

3.5 Scalability with increasing dataset sizes 48

3.6 Number of disk accesses 49

3.7 Scalability with increasing lengths 52

3.8 Scalability comparison (HDD) 53

3.9 Scalability comparison (SSD) 53

3.10 Exact methods footprint and TLB for synthetic datasets 57

3.11 Pruning ratio (Dataset Size= 100GB, Workload = 100 Queries) 58

3.12 Average time of queries with different difficulty (HDD) 61

3.13 Average time of queries with different difficulty (SSD) 62
3.14 Indexing and answering 100 queries (HDD) 63
3.15 Indexing and answering 100 queries (SSD) 64
3.16 Indexing and answering 10K queries (HDD) 65
3.17 Indexing and answering 10K queries (SSD) 66
3.18 Recommendations (Indexing and answering 10K queries on HDD) 71

4.1 Comparison of indexing scalability 89
4.2 Efficiency vs. accuracy for in memory ng-approximate search (100NN queries) 96
4.3 Efficiency vs. accuracy for in-memory δ-ϵ-approximate search(100NN queries) 97
4.4 Comparison of measures (Sift25GB) 98
4.5 Efficiency vs. accuracy for on disk ng-approximate search (100NN queries) 99
4.6 Efficiency vs. accuracy for on disk δ-ϵ-approximate search (100NN queries) 100
4.7 Efficiency vs. accuracy for the best methods ($\boldsymbol{\delta}$-$\boldsymbol{\epsilon}$-approximate) 101
4.8 Efficiency vs. k ($\boldsymbol{\epsilon}$-approximate) 102
4.9 Accuracy and efficiency vs. $\boldsymbol{\delta}$ and $\boldsymbol{\epsilon}$ 103
4.10 Recommendations (query answering). 108

5.1 The PAA and SAX representations 112
5.2 The APCA and EAPCA representations 113
5.3 A sample binary tree index created by the DSTree 114
5.4 DSTree Indexing Workflow 115
5.5 Hercules Indexing Workflow 117
5.6 Query Answering Workflow 121
5.7 Comparison of exact query answering using 1NN queries of different difficulty141
5.8 Exact query answering for best methods with increasing k 142
5.9 $\boldsymbol{\delta}$-$\boldsymbol{\epsilon}$-approximate search (Dataset= Synth250GB, Queries = 100NN) . . . 143

5.10 $\boldsymbol{\delta}$-$\boldsymbol{\epsilon}$-approximate search (Dataset= Deep250GB, Queries = 100NN) . . . 144
5.11 $\boldsymbol{\delta}$-$\boldsymbol{\epsilon}$-approximate search (Dataset= Seismic100GB, Queries = 100NN) . . 144
5.12 $\boldsymbol{\delta}$-$\boldsymbol{\epsilon}$-approximate search (Dataset= Sald100GB, Queries = 100NN) 145
5.13 $\boldsymbol{ng}$-approximate search (Dataset= Synth250GB, Queries = 100NN) . . . 147
5.14 $\boldsymbol{ng}$-approximate search (Dataset= Deep250GB, Queries = 100NN) 147
5.15 $\boldsymbol{ng}$-approximate search (Dataset= Seismic100GB, Queries = 100NN) . . 148
5.16 $\boldsymbol{ng}$-approximate search (Dataset= Sald100GB, Queries = 100NN) 148

List of Tables

3.1 Similarity search methods 37

3.2 Controlled workloads experimental results summary (sequential scan algorithm is highlighted) 56

4.1 Similarity search methods used in this study ("•" indicates our modifications to original methods). All methods support in-memory data, but only methods ticked in last column support disk-resident data. 75

Chapter 1

Introduction

We are surrounded by high-dimensional objects including data series, images, video, text, graphs, and deep neural network embeddings. Analyzing this data is important for a variety of real-world applications and has been extensively studied over the past 25 years. At the core of the analysis task lies a classic operation called similarity search.

The goal of this dissertation is to support efficient and accurate similarity search on massive collections of high-dimensional data. We present Hercules, a new technique for high-dimensional similarity search that outperforms the state-of-the-art. We also propose a taxonomy that unifies the nomenclature used for the different flavors of the similarity search problem, and design and execute two thorough experimental evaluations of the state-of-the-art exact and approximate similarity search approaches.

In this chapter, we first present a brief overview of the similarity search problem and the main data types we cover in our work (§ 1.1), then provide a succinct summary of the main contributions of this thesis (§ 1.2).

1.1 Problem Overview

We provide below a bird's eye view of the similarity search problem.

Similarity Search. Similarity search is a fundamental operation that lies at the core of many critical data processing tasks, such as data cleaning [1], data integration [2], and big data analytics (e.g., outlier detection [3, 4], frequent pattern mining [5], clustering [6, 7], and classification [8]). Similarity search aims at finding objects in a collection that are close to a given query according to some definition of sameness.

This problem has been studied heavily in the past 25 years and will continue to attract attention as massive collections of high-dimensional objects are becoming omnipresent in various domains. Objects can be data series, text, images, audio and video recordings, graphs, database tables or deep network embeddings (Fig. 1.1). Similarity search over high-dimensional objects is often reduced to a k-nearest neighbor (kNN) problem such that the objects are represented using high-dimensional vectors and the (dis)-similarity between them is measured using a distance. Some studies [9, 10, 11, 12] have argued that nearest neighbor search is not meaningful for a number of high-dimensional datasets due to the concentration of distances (a.k.a. the curse of dimensionality [13]), i.e., that the distance of a point to its farthest neighbor approaches the distance to its nearest neighbor. However, these conclusions were based on over-restrictive assumptions such as data being identical and independently distributed (i.i.d.) in each dimension, dimensionality being the only factor determining meaningfulness and an asymptotic analysis of dimensionality growing to infinity. In fact, other studies have shown that high-dimensional nearest neighbor search is meaningful for non-i.i.d data [9, 14, 15, 16], data with low intrinsic dimensionality and for a variety of real world datasets when meaningfulness is evaluated based on a number of data characteristics including finite dimensionality, data sparsity, dataset size and the nature of the distance measure used [17]. The importance and relevance of nearest neighbor search in high-dimensions is further evidenced by a large and growing body of research [18, 19, 20, 21, 22, 23, 24, 5, 25, 4, 26, 27, 28, 29, 30, 31,

32, 33, 34, 35, 36, 37, 38, 39].

This problem is hard because objects typically contain hundreds to thousands of dimensions and these dimensions need to be processed as a single entity not as individual values. Besides, as the dataset gets larger and cannot fit in-memory, the cost of the brute-force approach which compares a query to all objects in the collection becomes prohibitive both in terms of CPU and I/O. The research community has thus developed similarity search methods that aim at answering a query efficiently by limiting the number of data points accessed, while minimizing the I/O cost of accessing raw data on disk and the CPU cost when comparing raw data to the query (e.g., Euclidean distance calculations). These goals are achieved by exploiting summarization techniques, and using efficient data structures (e.g., an index) and search algorithms. Some methods further enhance the efficiency of similarity search by sacrificing accuracy returning deterministic or probabilistic approximate answers.

Answering a similarity query using an index typically involves two steps: a filtering step where the pre-built index is used to prune candidates and a refinement step where the surviving candidates are compared to the query in the original high-dimensional space [40, 41, 20, 28, 30, 42, 25, 43, 44, 31]. Some exact [41, 20, 43, 42] and approximate methods [26, 45] first summarize the original data and then index these summarizations, while others tie together data reduction and indexing [28, 30, 25]. Some approximate methods return the candidates obtained in the filtering step [45]. There also exist exact [46] and approximate [29] methods that index high-dimensional data directly. A variety of data structures exist for similarity search indexes, including trees [40, 43, 28, 30, 25, 44, 31, 26, 42], inverted indexes [47, 48, 49, 45], filter files [41, 20, 30], hash tables [50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60] and graphs [61, 62, 63, 64, 65, 66, 67, 29]. There also exist multi-step approaches, e.g., Stepwise [24], that transform and organize data according to a hierarchy of resolutions.

While our work is relevant to generic high-dimensional objects, we focus on two

prevalent types: data series and deep network embeddings. Both types of data exhibit correlated dimensions, a characteristic that has been shown to be favourable to meaningful nearest neighbor search [15].

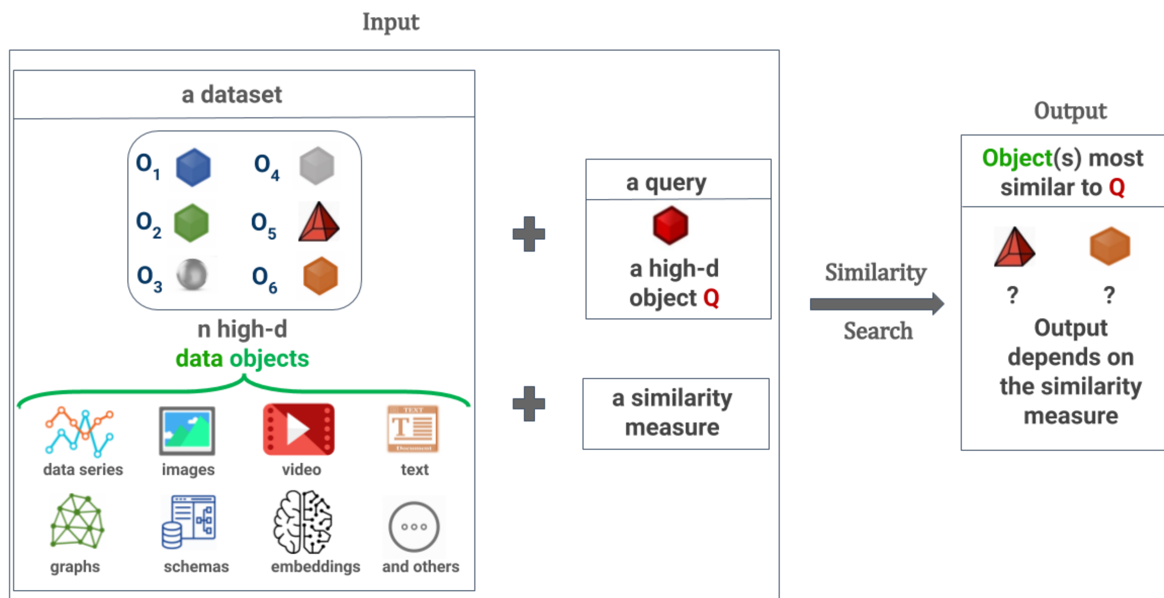


Figure 1.1: Similarity Search over Collections of High-Dimensional Objects

Data Series. A data series is a sequence of ordered real values. When the sequence is ordered on time, it is called a *time series*. However, the order can be defined by angle (e.g., in radial profiles), mass (e.g., in mass spectroscopy), position (e.g., in genome sequences), and others [68]. The terms *data series*, *time series* and *sequence* are used interchangeably. Data series are one of the most common types of data, covering virtually every scientific and social domain, such as astrophysics, neuroscience, seismology, environmental monitoring, biology, health care, energy, finance, criminology, social studies, video and audio recordings, and many others [69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82]. The number of data series generated by IoT technologies alone is estimated in multiple zettabytes [83]. A data series is typically represented as a high-dimensional vector of floating point values, where each value represents an observation and neighboring val-

ues are correlated. Data series similarity search is thus reduced to a k-nearest neighbor (kNN) query over the high-dimensional vectors of all series in a collection (Fig. 1.2).

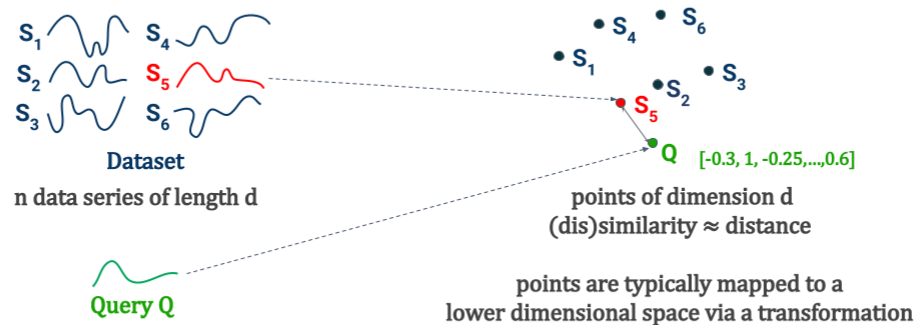


Figure 1.2: Data Series Similarity Search as a k-NN Query

Deep Network Embeddings. An embedding is a mapping of a high-dimensional object into a lower-dimensional vector such that similar objects are close together in the projected space. Embeddings are learned from the data using a deep neural network, where the embedding is a hidden layer and each dimension of the embedding is represented using one unit. A d-dimensional embedding is a vector of d floating point values, where each value is equal to the edge weight between the input node corresponding to the object and one of the d nodes in the embedding layer. Intuitively, each floating point value captures how important that particular facet (dimension) is for the input object. The number of dimensions d is a hyper-parameter of the learning model. Embeddings are used for three main tasks: 1) to support similarity search [84, 85]; 2) to serve as input to a machine learning model for a supervised task [86]; or 3) to enable the visualization of the relations between entities or concepts [87]. In our work, we are mainly interested in developing an efficient and scalable algorithm for scenario 1, i.e. similarity search. Embeddings have grown in popularity thanks to the recent breakthroughs in neural networks. They are used to represent various types of objects such as text [88, 85], graphs [89], images [90] and video [91]. Figure 1.3 shows a simplified architecture for a

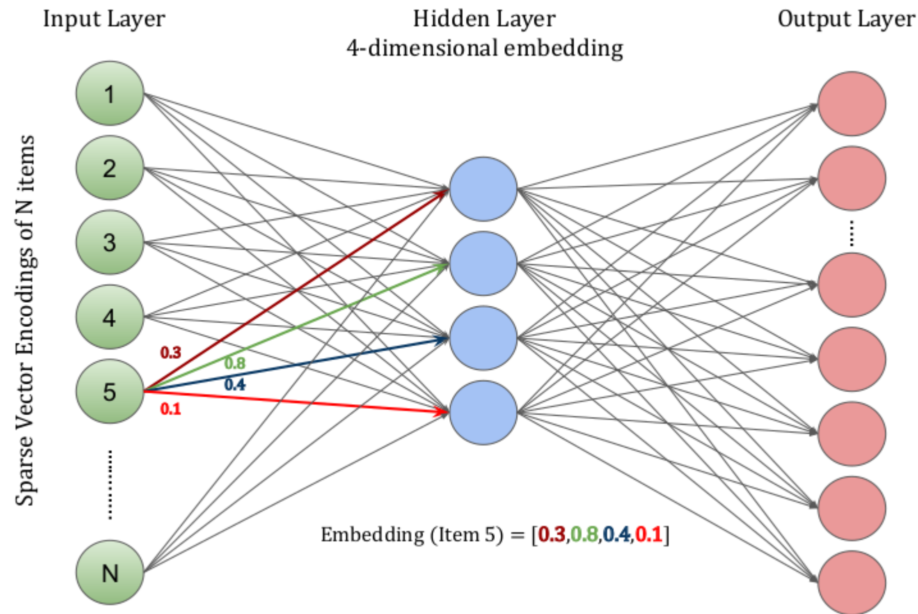


Figure 1.3: A simplified architecture for a neural network with a 4-dimensional embedding layer

deep neural network showing only the input and output layers and the 4-dimensional embedding layer. The input layer takes a sparse vector representation of a high-dimensional object, for instance a one-hot encoding of a word, then the model learns the weights of the edges connecting to the hidden layer as would occur in a supervised task, typically through back-propagation. Once the model is trained, the embedding vectors for all items in a collection can be used for other tasks, in our case similarity search.

1.2 Main Contributions

Our main contributions are as follows:

1. A unified terminology and taxonomy. We provide a framework for the similarity search nomenclature, unifying conflicting terminology from different research communities, and propose a taxonomy that classifies similarity search methods based on search quality guarantees.

2. An experimental evaluation of exact similarity search methods. We conducted the most extensive and comprehensive experimental evaluation in this area, including techniques from both the data series and high-dimensional data communities, which had never been considered together. We assess the efficiency and footprint of the different algorithms under a unified framework. We present an elaborate discussion and pinpointed promising research directions, including which approaches would benefit the most from modern hardware. We also demonstrate that choosing between an index or a scan is not a trivial decision but rather an optimization problem that depends on a variety of factors including hardware, query pruning ratio, data characteristics, the accuracy of a summarization and the efficacy of the clustering provided by an index. To help users decide the best approach for their problem, we issue a set of recommendations for typical user scenarios.

3. An experimental evaluation of approximate similarity search methods. In the approximate similarity search literature, experimental evaluations ignore the answering capabilities of data series methods. Our second experimental evaluation is the first study that fills this gap comparing the efficiency, accuracy and footprint of approximate data series approaches to state-of-the-art techniques designed for high-dimensional vectors. We pinpoint the weaknesses and the strengths of the different techniques sharing insights that have never been published in the literature. For instance, we show that the performance of popular LSH techniques [92] is inadequate for big data collections both in terms of efficiency and accuracy and that approximate techniques, that do not offer guarantees, are difficult to tune and can return incomplete results.

4. A new class of approximate similarity search techniques. We propose a new class of techniques for approximate similarity search that outperform the state-of-

the-art methods, including those designed for data series and generic high-dimensional vectors. We extended the best exact algorithms using ideas that were proposed 20 years ago for metric trees [18], and have not been exploited until now by any other technique. Our extensions are the clear winners for approximate similarity search with guarantees for both in-memory and disk based data, while they are the only viable solution for on-disk data. These techniques are also the fastest at indexing and have the lowest footprint. Another main advantage of our techniques is that the accuracy-efficiency trade-off is determined at query-time whereas all of the competitors need to perform this tuning both during index building and query answering, which means that an index may need to be built many times using different parameters before finding the right settings. Moreover, if the analyst builds an index with a particular accuracy target, and then their needs change, they will have to rebuild the index from scratch and go through the same process of determining the right parameter values. This becomes particularly impractical with larger datasets which can require several days to build an index.

5. A novel algorithm for exact and approximate similarity search. Building upon the deep insights gained from our experimental evaluations about the intricate designs of the different techniques, their strengths and weaknesses, we propose a new algorithm that outperforms our proposed extensions and the state-of-the-art exact and approximate similarity search techniques in-memory and on-disk. We used synthetic datasets and 4 of the largest publicly available real datasets of high-dimensional vectors, including data series, images, and deep neural network embeddings. Our datasets range in size from 25 gigabytes to 250 gigabytes. Similarity search algorithms based on indexes typically operate in two steps: a filtering step where the pre-built index is used to prune candidates and a refinement step where the surviving candidates are compared to the query in the original high dimensional space. Our new index owes its efficiency to the following novel ideas: 1) a double-filtering scheme which significantly reduces the num-

ber of surviving candidates compared to competing techniques; 2) an efficient storage mechanism for the index; 3) the carefully crafted distribution and execution of parallel instructions; and 4) the exploitation of the Single Instruction Multiple Data (SIMD) capabilities of modern CPUs.

6. A public archive which would serve as a the stepping stone for a much-needed benchmark. In both studies, all methods are evaluated under a unified framework to prevent implementation bias. We used the most efficient C/C++ implementations available for all approaches, and developed from scratch in C the ones that were only implemented in other programming languages, leading to new implementations that are considerably faster than the original ones. We share a public archive containing all source codes, datasets, queries and results [93, 94].

7. Far-reaching fundamental and practical implications. With the overwhelming data deluge that we have been witnessing in the past few years, the similarity search community has shown a growing interest in the development approximate methods following two main research trends: (i) LSH-based algorithms [26, 60] that support guarantees, but are relatively slow, and (ii) kNN graphs [45, 29] and inverted indexes [49, 47], which are relatively fast, but do not provide theoretical guarantees. We demonstrate that it is possible to design efficient high-dimensional vector similarity search algorithms with theoretical guarantees on the quality of the answers, and we thus offer a more promising alternative to the two current trends in the literature. This finding paves the way for very exciting new developments which will lead to efficient solutions that can support critical analytical tasks such as brain seizure detection, cyber-attack prevention, transportation management and data cleaning automation.

The contributions of this thesis have led to the following publications:

1. Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art. PVLDB, 12(2):112127, 2018. [95]

2. Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. Return of the Lernaean Hydra: Experimental Evaluation of Data Series Approximate Similarity Search. PVLDB, 13(3):402419, 2019. [96]

3. Karima Echihabi. Truly Scalable Data Series Similarity Search. In VLDB PhD Workshop, 2019. [97]

4. Karima Echihabi. High-Dimensional Similarity Search: From Time Series to Deep Network Embeddings. In SIGMOD, 2020. [98]

5. Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, Panagiota Fatourou, and Houda Benbrahim. Hercules: Overcoming the Lernaean Hydra of High Dimensional Similarity Search. Under Submission. [99]

6. Anna Gogolou, Theophanis Tsandilas, Karima Echihabi, Themis Palpanas, and Anastasia Bezerianos. Data Series Progressive Similarity Search with Probabilistic Quality Guarantees. In SIGMOD, 2020. [100]

1.3 Outline

The remainder of this thesis is organized in 5 chapters. Chapter 2 proposes a framework and taxonomy for the similarity search problem. Chapter 3 presents the results of the first comprehensive experimental evaluation of the efficiency of exact similarity search for high-dimensional vectors. Chapter 4 describes a new class of efficient approximate similarity search techniques for high-dimensional vectors and discusses the results of an exhaustive experimental study of approximate similarity search methods. Chapter

5 introduces Hercules, a new similarity search technique that outperforms the state-of-the-art methods in exact and approximate search. Chapter 6 concludes the thesis with a summary of the main findings, a statement of the contributions made by the research and an outline of promising future directions.

Chapter 2

Preliminaries

Increasingly large collections of high-dimensional objects are becoming commonplace across many different domains and applications. A key operation in the analysis of these collections is similarity search which aims at finding objects in a collection that are close to a given query according to some definition of sameness. This fundamental problem has attracted lots of attention and effort over the past two decades. Even though a large number of relevant approaches have been proposed in the literature by different communities, the non-standard use of terminology, with different terms being used for the same meaning and a single term being used with different meanings, makes it hard to compare and share results and often leads to confusion and misconceptions.

This chapter[1] is organized in two sections. In section 2.1, we provide definitions for the different flavors of similarity search that have been studied in the past, covering data types and distance measures (§ 2.1.1), similarity search queries (§ 2.1.2) and similarity search methods (§ 2.1.3). In section 2.2, we present a taxonomy that classifies similarity search methods based on the type of guarantees they provide.

[1]This chapter is a modified version of [95].

2.1 Definitions

2.1.1 Data Types and Distance Measures

In the context of similarity search, a complex object is often represented as a single point in an n-dimensional space, which is also referred to as a vector of n dimensions. An exhaustive description of the different object-to-vector mapping techniques is beyond the scope of this work, so for the sake of clarity, we only briefly describe how the data types we cover in this study are typically represented as vectors.

Images

An image is often represented with a *feature vector* [101], i.e., a vector of distinctive characteristics of this image, such as the color, texture or shape of a region of the image [101, 102] or a scale-invariant descriptor [103]. Each feature corresponds to a coordinate axis in the vector space.

Data Series

A ***data series*** $S(p_1, p_2, ..., p_n)$ is an ordered sequence of points, p_i, $1 \leq i \leq n$. The number of points, $|S| = n$, is the length of the series. We denote the i-th point in S by $S[i]$; then $S[i:j]$ denotes the ***subsequence*** $S(p_i, p_{i+1}, ..., p_{j-1}, p_j)$, where $1 \leq i \leq j \leq n$. We use $\mathbb{S}$ to represent all the series in a collection (dataset).

In the above definition, if each point in the series represents the value of a single variable (e.g., temperature) then each point is a scalar, and we talk about a ***univariate series***. Otherwise, if each point represents the values of multiple variables (e.g., temperature, humidity, pressure, etc.) then each point is a vector, and we talk about a ***multivariate series***. The values of a data series may also encode measurement errors, or imprecisions, in which case we talk about uncertain data series [104, 105, 106, 107, 4].

The scope of our work is univariate series with no uncertainty. In this case, a data

series S of length n is represented as a single point in an n-dimensional space by mapping each observation of the data series into one of the vector scalars. This type of vector is also called a *feature vector* and the values and length of S are referred to as *dimensions* and *dimensionality*, respectively.

From herein, unless otherwise specified, whenever we use the word data series or image, we refer to their feature vector representations.

Embeddings

An *embedding* is a mapping of an object into a point in low-dimensional real-valued vector space, called the *embedding space* [108]. The rationale is that the distances of the embedded space approximate those in the original space and that it is less expensive computationally to perform similarity search in the embedded space.

We are mainly concerned with embeddings that map *finite metric spaces* into *normed real-valued vector spaces.*

Consider the set $\mathbb{M}$ of finite cardinality N and the distance d, the tuple ($\mathbb{M}$,d) is said to be a ***finite metric space*** if $d : \mathbb{M} \times \mathbb{M} \rightarrow \mathbb{R}+$ is a distance metric.

A ***vector space*** $\mathbb{V}$ consists of a set of elements, called vectors, a field $\mathbb{F}$ of numbers, called scalars. and is closed under vector addition and scalar multiplication. The field $\mathbb{F}$ typically consists of the set of real numbers $\mathbb{R}$.

A ***normed real-valued vector space*** ($\mathbb{V}$,$\|.\|$) consists of the vector space $\mathbb{V}$ and the norm $\|.\|$. A norm is a real valued function with the following properties: $\forall\ u, v \in \mathbb{V}$ *and all* $a \in \mathbb{R}$

1. Triangular inequality: $\|u + v\| \leq \|u\| + \|v\|$
2. Absolute scalability: $\|av\| = |a|\|v\|$
3. Positivity: If $\|v\| = 0$ then v = 0 is the zero vector

While different norms have been studied in the literature, we use the Euclidean distance, which is a subclass of L_p norms where $p = 2$, because it is a popular choice as evidenced by a large body of work [18, 19, 20, 22, 23, 25, 4, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 109, 110, 97, 98].

Definition 1. *The **Euclidean distance** between two objects S_Q and S_C of dimensionality n, represented by two points $Q(q_1, q_2, ..., q_n)$ and $C(c_1, c_2, ..., c_n)$ in $\mathbb{R}^d$, is defined as follows [111]:*

$$d(S_Q, S_C) \equiv d(Q, C) = \sqrt[2]{\sum_{i=1}^{n} (q_i - c_i)^2}$$

Definition 2. *An **embedding** f that maps an object from a finite metric space ($\mathbb{M}$,d) to a normed real-valued vector space ($\mathbb{V}$,d') is defined as $f : \mathbb{M} \rightarrow \mathbb{R}^m$, where m is the dimensionality of the embedding space and d' is the distance metric in the embedding space [108].*

An embedding is called ***isometric*** if the distances in the embedding space are equal to their correponding distances in the original space. An embedding is called ***contractive*** if the distances in the embedding space lower-bound the corresponding distances in the original space.

Embeddings have been studied extensively in pure mathematics [112, 113, 114, 115] and have been exploited to model different types objects including graphs [116, 117], images [118], protein sequences [119], and data series [120, 121]. The popularity of embeddings has surged in the past few years thanks to the machine learning revolution [88, 85, 89, 122].

When the object is a high-dimensional vector, we call this special class of embeddings, *dimensionality reduction* techniques [108]. Examples of dimensionality reduction techniques are the Karhunen-Loève transform (KLT) [123], the Discrete Fourier Transform [120, 124] and wavelets [121]. We will provide a more detailed survey of these techniques in Chapters 3 and 4 .

For instance, for a data series representing an audio signal, an embedding can be extracted from the high-dimensional power spectral density or Fourier coefficients. Similarly, an embedding can be mapped from the one-hot-encoding of a word, or the raw pixel intensities of an image or its SIFT descriptors [125].

In the literature, the terms *embeddings* and *feature vectors* are sometimes used interchangeably. In this thesis, we use the term *embedding* to refer to a a low-dimensional and dense vector representation. We call embeddings created by deep neural networks *deep network embeddings* and those created by other techniques simply *embeddings*, in particular, vectors resulting from a dimensionality reduction technique. Note that an embedding does not always represent a feature vector of an individual object but can also model input interaction context, the most popular example being recommender systems based on collaborative filtering. For example, the embeddings created by movie recommendation systems model the users identifiers and the list of movies they watched [126]. Besides, embeddings can model multimodal entities to enable translation from one modality to another, for instance to automatically generate captions for images [127].

When similarity search is reduced to a k-NN problem, the high-dimensional objects are represented as n-dimensional vectors, and the (dis)-similarity between the objects is evaluated as the *distance* between their vector representations. The distance between a query object, S_Q, and a candidate object, S_C, is denoted by $d(S_Q, S_C)$. Even though several distance measures have been proposed in the literature [128, 129, 130, 131, 132, 133], the Euclidean distance is the one that is the most widely used, as well as one of the most effective for large data series collections [134]. In the case of data series, we note that an additional advantage of the Euclidean distance is that when the series is Z-normalized (mean=0, stddev=1), which is very often done in practice [135], it can be exploited to compute Pearson correlation [136].

Since the Euclidean distance is computationally expensive, for similarity search methods to be efficient, they need to reduce the number of these calculations. Some of these

methods achieve this by creating embeddings for the high-dimensional feature vectors of all objects in the dataset, then performing the distance calculations in the embedding space. If the embedding is isometric, the result in the embedding space is guaranteed to be correct in the original space. However, embeddings are rarely isometric, so similarity search techniques often exploit contractive embeddings so they can use the search in the embedding space as a filtering step, then further refine the filtered candidates by comparing their distances in the original high-dimensional space. These techniques rely on *lower-bounding* distances [28, 30, 42, 25, 44, 31, 46, 24] and some of them also define *upper-bounding* distances [25, 24].

Definition 3. *Consider an embedding f from ($\mathbb{M}$,d) to ($\mathbb{V}$,d'), where d and d' are the distance metrics in the original and embedding spaces respectively, the distance d' is a* ***lower-bounding*** *distance if:* $\forall\ S_Q, S_C \in \mathbb{M}\ \ d'(f(S_Q), f(S_C)) \leq d(S_Q, S_C)$ *[124].*

Definition 4. *Consider an embedding f from ($\mathbb{M}$,d) to ($\mathbb{V}$,d'), where d and d' are the distance metrics in the original and embedding spaces respectively, the distance d' is an* ***upper-bounding*** *distance if:* $\forall\ S_Q, S_C \in \mathbb{M}\ \ d'(f(S_Q), f(S_C)) \geq d(S_Q, S_C)$ *[25, 24].*

2.1.2 Similarity Search Queries

We now define the different forms of similarity search queries. We assume a data collection of objects, $\mathbb{S}$, a query object, S_Q, and a distance function $d(\cdot, \cdot)$.

A ***k-Nearest-Neighbor (k-NN) query*** identifies the k objects in the collection with the smallest distances to the query.

Definition 5. *Given an integer k, a* ***k-NN query*** *retrieves the set of objects* $\mathbb{A} = \{\{S_{C_1}, ..., S_{C_k}\} \subseteq \mathbb{S} | \forall\ S_C \in \mathbb{A}$ *and* $\forall\ S_{C'} \notin \mathbb{A},\ d(S_Q, S_C) \leq d(S_Q, S_{C'})\}$.

An ***r-range query*** identifies all the series in the collection within range r form the query series.

Definition 6. *Given a distance* r*, an* ***r-range query*** *retrieves the set of objects* $\mathbb{A} = \{S_C \in \mathbb{S} | d(S_Q, S_C) \leq r\}$*.*

In the case of data series, we additionally identify the following two categories of k-NN and range queries. In ***whole matching (WM)*** queries, we compute the similarity between an entire query series and an entire candidate series. All the series involved in the similarity search should to have the same length. In ***subsequence matching (SM)*** queries, we compute the similarity between an entire query series and all subsequences of a candidate series. In this case, candidate series can have different lengths, but should be longer than the query series.

Definition 7. *A* ***whole matching query*** *finds the candidate data series* $S \in \mathbb{S}$ *that matches* S_Q*, where* $|S| = |S_Q|$*.*

Definition 8. *A* ***subsequence matching query*** *finds the subsequence* $S[i:j]$ *of a candidate data series* $S \in \mathbb{S}$ *that matches* S_Q*, where* $|S[i:j]| = |S_Q| < |S|$*.*

In practice, we encounter situations that cover the entire spectrum: WM queries on large collections of short series [137, 138], SM queries on large collections of short series [139], and SM queries on collections of long series [140].

Note that SM queries can be converted to WM: create a new collection that comprises all overlapping subsequences (each long series in the candidate set is chopped into overlapping subsequences of the length of the query), and perform a WM query against these subsequences [31, 141].

In this work, we focus on *whole-matching* queries. This is a very popular problem that lies at the core of several other algorithms, and is important for many applications in various domains in the real world [7, 135, 68], ranging from fMRI clustering [142] to mining earthquake [143], energy consumption [144], and retail data [145].

2.1.3 Similarity Search Methods

When a similarity search algorithm (k-NN or range) produces answers that are (by definition) always correct and complete: we call such an algorithm ***exact***. Nevertheless, we can also develop algorithms without such strong guarantees: we call such algorithms ***approximate***. As we discuss below, there exist different flavors of approximate similarity search algorithms.

An **ϵ-approximate** algorithm guarantees that its distance results have a relative error no more than ϵ, i.e., the approximate distance is at most $(1 + \epsilon)$ times the exact one.

Definition 9. *Given a query S_Q, and $\epsilon \geq 0$, an **ϵ-approximate** algorithm guarantees that all results, S_C, are at a distance $d(S_Q, S_C) \leq (1 + \epsilon)d(S_Q, [\text{k-th NN of } S_Q])$ in the case of a k-NN query, and distance $d(S_Q, S_C) \leq (1 + \epsilon)r$ in the case of an r-range query.*

A **δ-ϵ-approximate** algorithm, guarantees that its distance results will have a relative error no more than ϵ (i.e., the approximate distance is at most $(1 + \epsilon)$ times the exact distance), with a probability of at least δ.

Definition 10. *Given a query S_Q, $\epsilon \geq 0$, and $\delta \in [0, 1]$, a **δ-ϵ-approximate** algorithm produces results, S_C, for which $Pr[d(S_Q, S_C) \leq (1 + \epsilon)d(S_Q, [\text{k-th NN of } S_Q])] \geq \delta$ in the case of a k-NN query, and $Pr[d(S_Q, S_C) \leq (1 + \epsilon)r] \geq \delta$) in the case of an r-range query.*

An ***ng-approximate*** (no-guarantees approximate) algorithm does not provide any guarantees (deterministic, or probabilistic) on the error bounds of its distance results.

Definition 11. *Given a query S_Q, an **ng-approximate** algorithm produces results, S_C, that are at a distance $d(S_Q, S_C) \leq (1 + \theta)d(S_Q, [\text{k-th NN of } S_Q])$ in the case of a k-NN query, and distance $d(S_Q, S_C) \leq (1 + \theta)r$ in the case of an r-range query, for an arbitrary value $\theta \in \mathbb{R}_{>0}$.*

In the data series literature, *ng-approximate* algorithms have been referred to as *approximate*, or *heuristic* search [28, 30, 42, 25, 44, 31]. Unless otherwise specified, for

the rest of this paper we will refer to *ng-approximate* algorithms simply as approximate. Approximate matching in the data series literature consists of pruning the search space, by traversing one path of an index structure representing the data, visiting at most one leaf, to get a baseline best-so-far (bsf) match.

Observe that when $\delta = 1$, a δ-ϵ-approximate method becomes ϵ-approximate, and when $\epsilon = 0$, an ϵ-approximate method becomes exact [18]. It it also possible that the same approach implements both approximate and exact algorithms [146, 25, 28, 30, 42]. Methods that provide exact answers with probabilistic guarantees are considered δ-0-approximate. These methods guarantee distance results to be exact with probability at least δ ($0 \leq \delta \leq 1$ and $\epsilon = 0$). (We note that in the case of k-NN queries, Def. 9 corresponds to the *approximately correct NN* [18] and $(1 + \epsilon)$-*approximate NN* [147], while Def. 10 corresponds to the *probably approximately correct NN* [18].)

Figure 2.1 shows a visual interpretation of these different flavors of similarity search for $(\mathbb{R}^2, L_2)$. The radius $r_\delta(O_Q)$ is the maximum distance from O_Q, such that the sphere with center O_Q and radius $r_\delta(O_Q)$ is empty with probability δ.

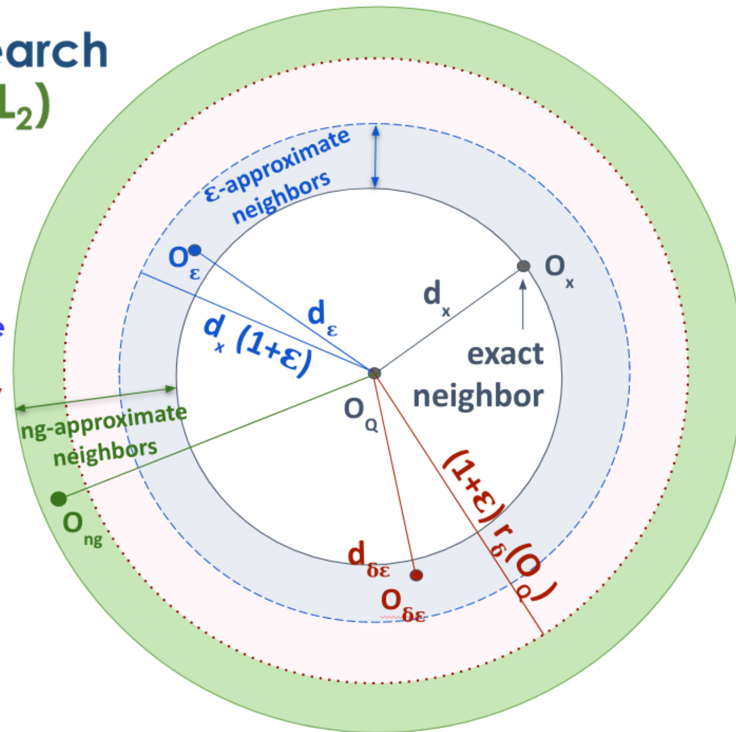


Figure 2.1: Different flavors of the similarity search problem

2.2 A Taxonomy

Figure 2.2 presents a taxonomy of similarity search methods based on the type of guarantees they provide as defined in the previous section (methods with multiple types of guarantees are included in more than one leaf of the taxonomy). We call probabilistic the general δ-ϵ-approximate methods. When $\delta = 1$ we have the ϵ-approximate methods. Setting $\delta = 1$ and $\epsilon = 0$, we get the exact methods. Finally, methods that provide no guarantees are categorized under ng-approximate.

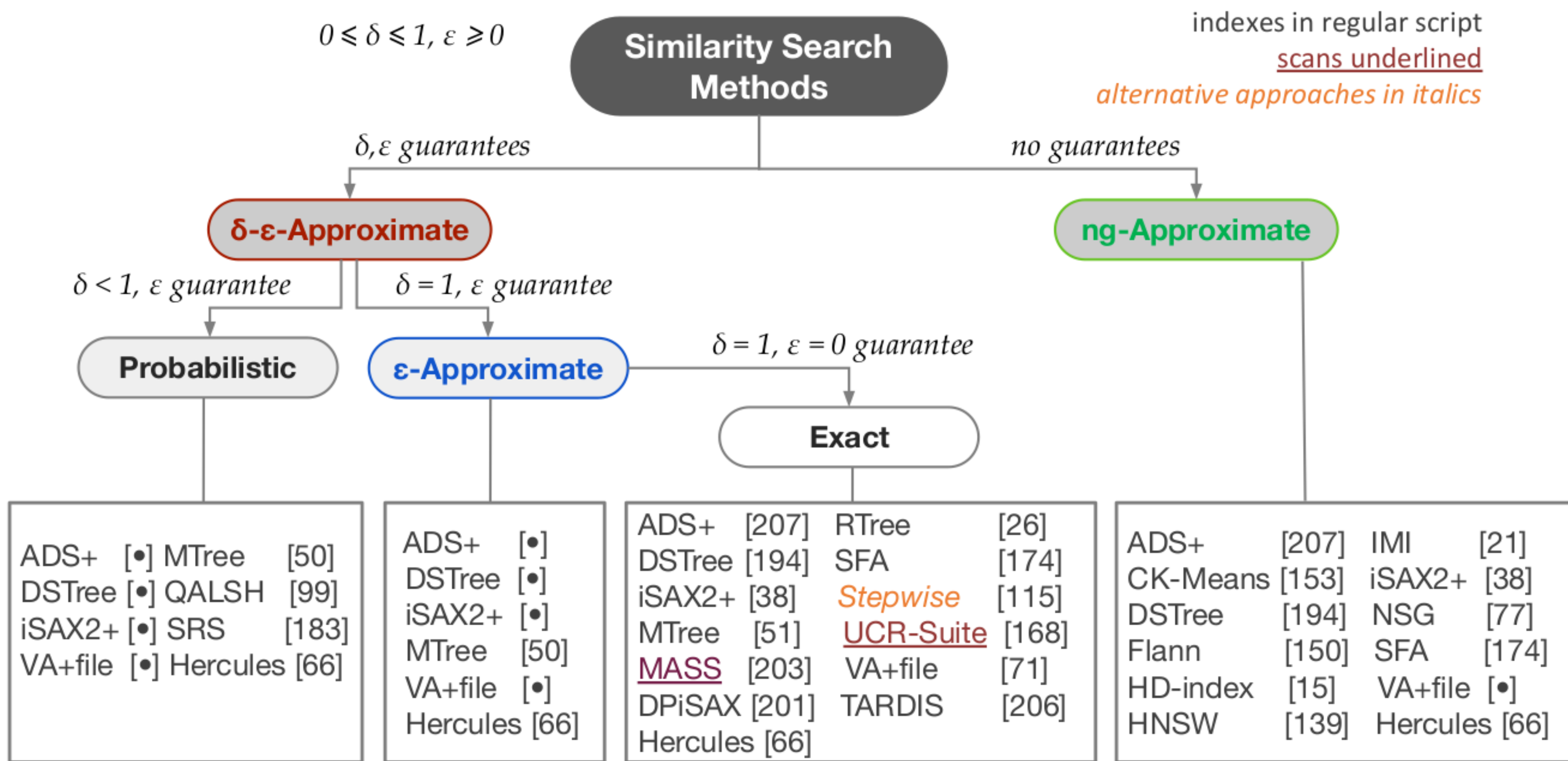


Figure 2.2: Taxonomy of similarity search methods
(• indicates our extensions to existing techniques)

2.3 Conclusion

In the following chapters, we will provide a detailed description of all the methods referenced in the taxonomy. Chapter 3 will study exact techniques, Chapter 4 will examine probabilistic, ϵ-approximate and ng-approximate methods, including the new class of techniques that we propose for similarity search, and Chapter 5 will cover our novel similarity search algorithm.

Chapter 3

Exact Similarity Search

Similarity search is a core operation of many critical data processing tasks, such as data cleaning, data integration and data analytics (e.g., outlier detection, frequent pattern mining, clustering and classification). Similarity search algorithms can either return exact or approximate answers. Exact methods are expensive while approximate methods sacrifice accuracy to achieve better efficiency. Although several exact approaches have been proposed in the literature in the past 25 years, none of the existing studies provides a detailed evaluation against the available alternatives.

In this chapter[1], we present the first systematic experimental evaluation of the efficiency of exact similarity search techniques for high-dimensional vectors. Based on the experimental results, we describe the strengths and weaknesses of each approach and give recommendations for the best approach to use under typical use cases. Finally, by identifying the shortcomings of each method, our findings lay the ground for solid further developments in the field.

The chapter is organized in six sections. We summarize our contributions in section 3.1, briefly survey the state-of-the-art exact techniques in section 3.2, describe our experimental framework in section 3.3, present results and a thorough discussion in sections 3.4 and 3.5, then close the chapter with concluding remarks in section 3.6.

[1]This chapter is a slightly modified version of [95].

3.1 Main Contributions

Our main contributions are as follows:

1. We include a brief survey of similarity search approaches, bringing together studies presented in different communities that have been treated in isolation from each other. These approaches range from smart serial scan methods to the use of indexing, and are based on a variety of classic and specialized data summarization techniques, including both traditional signal processing techniques and modern specialized ones.

2. We make sure that all approaches are evaluated under the same conditions, so as to guard against implementation bias. To this effect, we used implementations in C/C++ for all approaches, and reimplemented in C the ones that were only available in other programming languages. Moreover, we conducted a careful inspection of the code bases, and applied to all of them the same set of optimizations (e.g., with respect to memory management, Euclidean distance calculation, etc.), leading to considerably faster performance.

3. We conduct the first comprehensive experimental evaluation of the efficiency of similarity search approaches including techniques designed for metric spaces, multidimensional data, and data series. We use several synthetic and 4 real datasets from diverse domains. In addition, we report the first large scale experiments with carefully crafted query workloads that include queries of varying difficulty, which can effectively stress-test all the approaches. Our results reveal characteristics that have not been reported in the literature, and lead to a deep understanding of the different approaches and their performance. Based on those, we provide recommendations for the best approach to use under typical use cases, and identify promising future research directions.

4. We make available online all source codes, datasets, and query workloads used in our study [93]. This will render our work reproducible and further help the community to agree on and establish a much needed high-dimensional similarity search benchmark [148, 135, 149].

3.2 Approaches

Similarity search methods can be classified into sequential, and indexing methods. Sequential methods proceed in one step to answer a similarity search query. Each candidate is read sequentially from the raw data file and compared to the query. Particular optimizations can be applied to limit the number of these comparisons [5]. Some sequential methods work with the raw data in its original high-dimensional representation [5], while others perform transformations on the raw data before comparing them to the query [27].

On the other hand, answering a similarity query using an index involves two steps: a filtering step where the pre-built index is used to prune candidates and a refinement step where the surviving candidates are compared to the query in the original high dimensional space [40, 41, 20, 28, 30, 42, 25, 43, 44, 31, 141]. Some indexing methods first summarize the original data and then index these summarizations [43, 42, 41, 20], while others interwine data reduction and indexing [28, 30, 25]. Some methods index high dimensional data directly [46]. We note that all indexing methods depend on lower-bounding, since it allows indexes to prune the search space with the guarantee of no false dismissals [124] (the DSTree index [25] also supports an upper-bounding distance, but does not use it for similarity search). Metric indexes (such as the M-tree [46]) additionally require the distance measure triangle inequality to hold. Though, there exist (non-metric) indexes for data series that are based on distance measures that are not metrics [150].

There also exist hybrid approaches that fall in-between indexing and sequential methods. In particular, multi-step approaches, where data is transformed and re-organized in levels. Pruning then occurs at multiple intermediate filtering steps as levels are sequentially read one at a time.

Stepwise is such a method [24], relying on the Euclidean distance, and lower- and upper-bounding distances.

3.2.1 Summarization Techniques

We now briefly outline the summarization techniques used by the methods that we examine in this study.

The *Discrete Haar Wavelet Transform* (DHWT) [121] uses the Haar wavelet decomposition to transform each object S into a multi-level hierarchical structure. Resulting summarizations are composed of the first l coefficients.

The *Discrete Fourier Transform* (DFT) [120, 124, 151, 152] decomposes S into frequency coefficients. A subset of l coefficients constitutes the summary of S. In our experiments, we use the Fast Fourier Transform (FFT) algorithm, which is optimal for whole matching scenarios (the MFT algorithm [153] is faster than FFT for computing DFT on sliding windows, thus beneficial for subsequence matching queries).

The *Piecewise Aggregate Approximation* (PAA) [154] and *Adaptive Piecewise Constant Approximation* (APCA) [155] methods are segmentation techniques that divide S into l (equi-length and varying-length, respectively) segments. Each segment represents the mean value of the corresponding points. The *Extended Adaptive Piecewise Approximation* (EAPCA) [25] technique extends APCA by using more information to represent each segment. In addition to the mean, it also stores the standard deviation of the segment. With the *Symbolic Aggregate Approximation* (SAX) [156], S is first transformed using PAA into l real values, and then a discretization technique is applied to map PAA values to discrete set of symbols (alphabet) that can be succinctly represented in binary form. A SAX representation consists of l such symbols. An iSAX (indexable SAX) [157] representation can have an arbitrary alphabet size for each segment.

Similarly to SAX, the *Symbolic Fourier Approximation* (SFA) [42] is also a symbolic approach. However, instead of PAA, it first transforms S into l DFT coefficients using FFT (or MFT for subsequence matching), then extends the discretization principle of SAX to support both equi-depth and equi-width binning, and to allow each dimension to have its own breakpoints. An SFA summary consists of l symbols.

Using the VA+file method [20], an object S of dimension n is first transformed using the Karhunen–Loève transform (KLT) into n real values, which are then quantized to discrete symbols. As we will detail later, we modified the VA+file to use DFT instead of KLT, for efficiency reasons.

Figure 3.1 presents a high-level overview of the summarization techniques presented above.

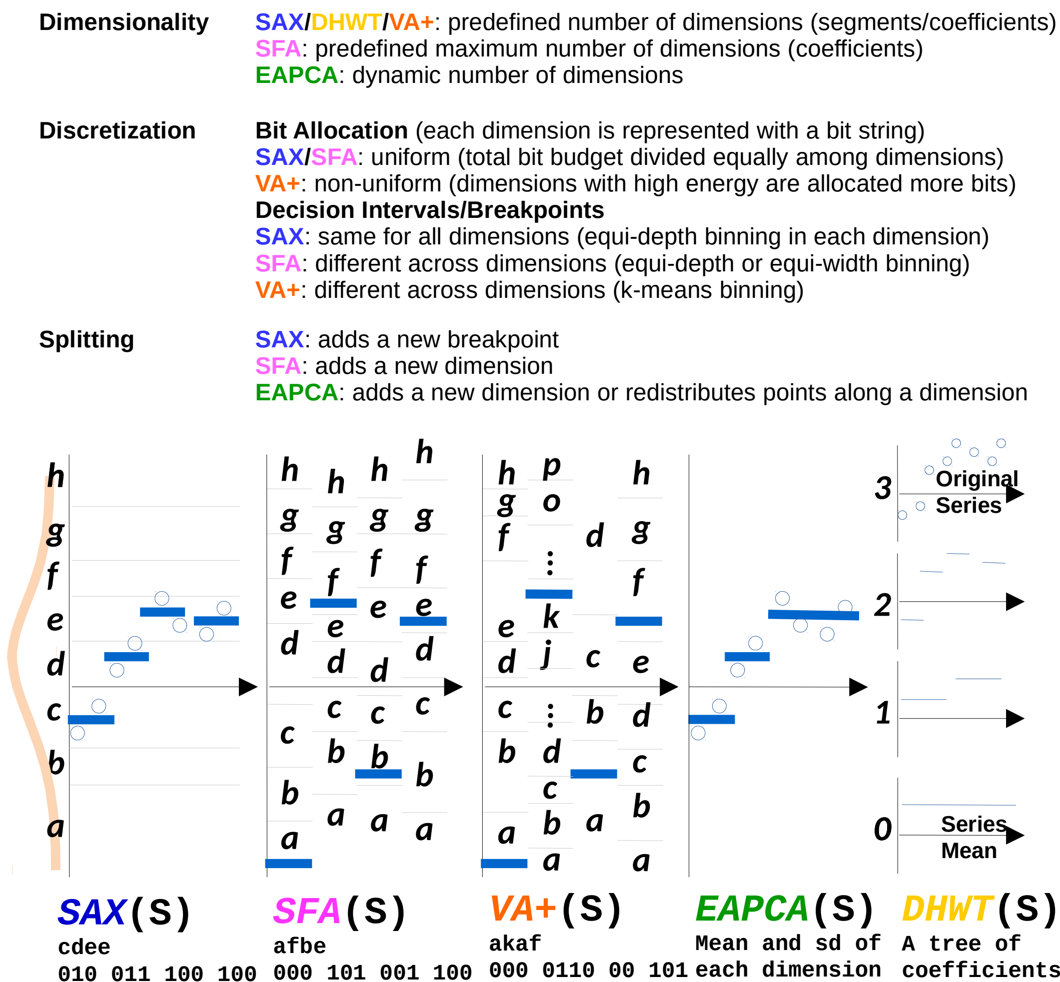


Figure 3.1: Summarizations

3.2.2 Similarity Search Methods

In this study, we focus on algorithms that can produce exact results, and evaluate the ten methods outlined below (in chronological order). The properties of these algorithms are also summarized in Table 3.1.

We also point out that there exist several techniques dedicated to approximate similarity search [158, 21, 26, 159, 29, 90]. A thorough evaluation of all approximate methods is covered in Chapter 4.

R*-tree. The R*-tree [43] is a height-balanced spatial access method that partitions the data space into a hierarchy of nested overlapping rectangles. Each leaf can contain either the raw data objects or pointers to those, along with the enclosing rectangle. Each intermediate node contains the minimum bounding rectangle that encompasses the rectangles of its children. Given a query S_Q, the R*-tree query answering algorithm visits all nodes whose rectangle intersects S_Q, starting from the root. Once a leaf is reached, all its data entries are returned. We tried multiple implementations of the R*-tree, and opted for the fastest [160]. We modified this code by adding support for PAA summaries.

M-tree. The M-tree [46] is a multidimensional, metric-space access method that uses hyper-spheres to divide the data entries according to their relative distances. The leaves store data objects, and the internal nodes store routing objects; both store distances from each object to its parent. During query answering, the M-tree uses these distances to prune the search space. The triangle inequality that holds for metric distance functions guarantees correctness. Apart from exact queries, it also supports ϵ-approximate and δ-ϵ-approximate queries. We experimented with four different code bases: two implementations that support bulk-loading [161, 4], the disk-aware mvptree [162], and a memory-resident implementation [4]. We report the results with the latter, because (despite our laborious efforts) it was the only one that scaled to datasets larger than 1GB. We modified it to use the same sampling technique as the original implementation [161], which chooses the number of initial samples based on the leaf size, minimum utilization,

and dataset size.

VA+file. The VA+file [20] is an improvement of the VA-file method [41]. While both methods create a filter file containing quantization-based approximations of the high dimensional data, and share the same exact search algorithm, the VA+file does not assume that neighboring points (dimensions) in the sequence are uncorrelated. It thus improves the accuracy of the approximations by allocating bits per dimension in a non-uniform fashion, and partitioning each dimension using a k-means (instead of an equi-depth approach). We improved the efficiency of the original VA+file significantly by implementing it in C and modifying it to use DFT instead of KLT, since DFT is a very good approximation for KLT [20] and is much more efficient [163].

Stepwise. The Stepwise method [24] differentiates itself from indexing methods by storing DHWT summarizations vertically across multiple levels. This process happens in a pre-processing step. When a query S_Q arrives, the algorithm converts it to DWHT, and computes the distance between S_Q and the DHWT of each candidate data series one level at a time, using lower and upper bounding distances it filters out non-promising candidates. When leaves are reached, the final refinement step consists of calculating the Euclidean distance between the raw representations of S_Q and the candidate series. We modified the original implementation to load the pre-computed sums in memory and answer one query at a time (instead of the batch query answering of the original implementation). We also slightly improved memory management to address swapping issues that occurred with the out-of-memory datasets.

SFA trie. The SFA approach [42] first summarizes the series using SFA of length 1 and builds a trie with a fanout equal to the alphabet size on top of them. As leaves reach their capacity and split, the length of the SFA word for each series in the leaf is increased by one, and the series are redistributed among the new nodes. The maximum resolution is the number of DFT coefficients given as a parameter. SFA implements lower-bounding to prune the search space, as well as a bulk-loading algorithm. We re-

implemented SFA in C, optimized its memory management, and improved the sampling and buffering schemes. This resulted in a significantly faster implementation than the original one in Java.

UCR Suite. The UCR Suite [5] is an optimized sequential scan algorithm for exact subsequence matching. We adapted the original algorithm to support exact whole matching.

DSTree. The DSTree [25] approach uses the EAPCA summarization technique, which allows, during node splitting, the resolution of a summarization to increase along two dimensions: vertically and horizontally. (Instead, SAX-based indexes allow horizontal splitting by adding a breakpoint to the y-axis, and SFA allows vertical splitting by adding a new DFT coefficient.) In addition to a lower bounding distance, the DSTree also supports an upper bounding distance. It uses both distances to determine the optimal splitting policy for each node. We reimplemented the DSTree algorithm in C and we optimized its buffering and memory management, improving the performance of the algorithm by a factor of 4, compared to the original implementation (in Java).

iSAX2+. The iSAX family of indexes has undergone several improvements [164]. The iSAX 2.0 index [23] improved the splitting policy and added bulk-loading support to the original iSAX index [146]. iSAX2+ [28] further optimized bulk-loading. In the literature, competing approaches have either compared to iSAX, or iSAX 2.0. This is the first time that iSAX2+ is compared to other exact data series indexes. The index supports ng-approximate and exact query answering. We reimplemented the original iSAX2+ algorithm from scratch using C, and optimized its memory management, leading to significant performance improvements.

ADS+. ADS+ [30] is the first query adaptive data series index. It first builds an index tree structure using only the iSAX summarizations of the raw data, and then adaptively constructs the leaves and incorporates the raw data during query answering. For exact query answering, the SIMS algorithm is proposed. It first performs a fast ng-approximate search in the tree in order to acquire an initial best-so-far (bsf) distance, then prunes

the search space by using the bsf and the lower bounds between the query and all iSAX summaries. Using that, it performs a skip-sequential search on the raw data that were not pruned. In all our experiments involving ADS+ we use the SIMS algorithm for exact similarity search. ADS-FULL is a non-adaptive version of ADS, that builds a full index using a double pass on the data.

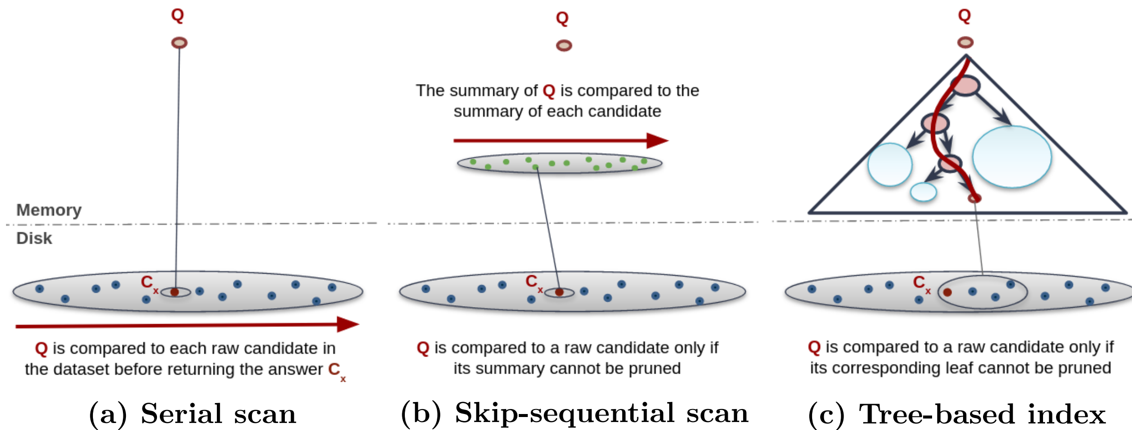
MASS. MASS [165] is an exact subsequence matching algorithm, which computes the distance between a query, S_Q, and every subsequence in the series, using the dot product of the DFT transforms of the series and the reverse of S_Q. We adapted it to perform exact whole matching queries.

Table 3.1: Similarity search methods

		Matching Accuracy				Matching Type		Representation		Implementation	
		exact	ng-appr.	ϵ-appr.	δ-ϵ-appr.	Whole	Subseq.	Raw	Reduced	Original	New
Indexes	ADS+	[30]	[30]			✓			iSAX	C	
	DSTree	[25]	[25]			✓			EAPCA	Java	C
	iSAX2+	[28]	[28]			✓			iSAX	C#	C
	M-tree	[46]		[18]	[18]	✓		✓		C++	
	R*-tree	[43]				✓			PAA	C++	
	SFA trie	[42]	[42]			✓	✓		SFA	Java	C
	VA+file	[20]				✓			DFT	MATLAB	C
Other	UCR Suite	[5]					✓	✓		C	
	MASS	[165]					✓		DFT	C	
	Stepwise	[24]				✓			DHWT	C	

3.2.3 Examples

Figure 3.2 illustrates how a serial scan (e.g., the UCR-Suite), a skip-sequential scan (e.g., ADS+ and the VA+file) and a tree-based index (e.g., iSAX2+, DSTree and SFA) typically answer a similarity search query.



(a) Serial scan **(b) Skip-sequential scan** **(c) Tree-based index**

Figure 3.2: Answering a similarity search query using different access paths

A similarity search query Q is answered using a serial scan by comparing it to each single candidate in the high-dimensional space (Fig. 3.2a). This incurs a high CPU cost due to the large number of distance calculations. Moreover, when the dataset is too large to fit in memory, the overhead for reading all the high-dimensional data from disk becomes prohibitive.

A skip-sequential scan typically performs first a search in-memory on the summarized data, then compares the query to the raw candidates that could not be pruned based on their summaries (Fig. 3.2b). The key principle that supports pruning is the lower-bounding property, i.e., distances in the reduced dimensionality space are guaranteed to be smaller than or equal to distances in the original space (Definition 3 in Chapter 2). Before search starts, the best-so-far distance (bsf) is initialized to $+\infty$. When search starts, the lower-bounding distance (lb_1) is calculated between the summary of Q and the summary of the first candidate C_1. Since lb_1 is smaller than bsf, C_1 cannot be pruned; therefore the distance between Q and C_1 is calculated ($real_1$). Since $real_1$ is smaller

than bsf, the latter is updated with $real_1$. Search proceeds with the second candidate calculating lb_2, the lower-bounding distance between the summary of Q and the summary of C_2. If lb_2 is greater than or equal to bsf, C_2 can be pruned because the lower-bounding property guarantees that $real_2 \geq lb_2 \geq bsf$; therefore C_2 will not improve the bsf. If lb_2 is smaller than bsf, C_2 cannot be pruned and the algorithm calculates $real_2$. The bsf is updated with $real_2$ if and only if $real_2 < bsf$. Search continues similarly until all summaries are processed. Different summarizations have been used in the literature, for example ADS+ uses SAX summaries and the VA+file uses DFT transformations. The summaries and the raw data are stored in the same order.

A tree-based index also takes advantage of lower-bounding to reduce the number of distance calculations between the query and the candidates in the dataset (Fig. 3.2c). Typically, each internal node in the tree summarizes the data contained in its subtree and each leaf node is associated with a file on disk that contains the raw high-dimensional data. Raw data is loaded from disk only if the leaf containing it could not be pruned. When a similarity search query arrives, a quick approximate answer is returned by heuristically visiting one leaf, loading the leaf's data into disk, calculating the distance between the query and all the candidates in this leaf and initializing the bsf with the smallest distance. Search proceeds by calculating lb_1, the distance between the root summarization and the query summarization. Following the same lower-bounding principle described earlier, if $lb_1 \geq bsf$, the root can be pruned because no other candidate will improve the bsf, and search terminates by returning bsf as the final answer. However, if $lb_1 < bsf$, then the root cannot be pruned, and its children are visited. Search proceeds in the same fashion until all nodes have either been pruned or visited. When a leaf is visited, its raw data is loaded into memory and distances are calculated between the query and all the leaf's raw data. Any such distance that is smaller than the current bsf becomes the new bsf. When all nodes in the tree have been processed, search returns the current bsf as the final answer.

3.3 Experimental Evaluation

In order to provide an unbiased evaluation, we re-implemented in C all methods whose original language was other than C/C++. Our new implementations are more efficient (in space and time) than the original ones on all datasets we tested. All methods use single precision values, and the methods based on fixed summarizations use 16 segments/coefficients. The same set of known optimizations for data series processing are applied to all methods. All results, source codes, datasets and plots are available in [93].

3.3.1 Environment

All methods were compiled with GCC 6.2.0 under Ubuntu Linux 16.04.2 with level 2 optimization. Experiments were run on two different machines. The first machine, called *HDD*, is a server with two Intel Xeon E5-2650 v4 2.2GHz CPUs, 75GB[2] of RAM, and 10.8TB (6 x 1.8TB) 10K RPM SAS hard drives in RAID0. The throughput of the RAID0 array is 1290 MB/sec. The second machine, called *SSD*, is a server with two Intel Xeon E5-2650 v4 2.2Ghz CPUs, 75GB of RAM, and 3.2TB (2 x 1.6TB) SATA2 SSD in RAID0. The throughput of the RAID0 array is 330 MB/sec. All our algorithms are single-core implementations.

3.3.2 Experimental Setup

Scope. This work concentrates on exact whole-matching (WM) 1-NN queries. Extending our experimental framework to cover r-range queries, subsequence matching and approximate query answering is part of our future work.

Algorithms. This experimental study covers the ten methods described in Section 3.2,

[2]We used GRUB to limit the amount of RAM, so that all methods are forced to use the disk. Note that GRUB prevents the operating system from using the rest of the RAM as a file cache, which is what we wanted for our experiments.

which all have native-support for Euclidean distance. Our baseline is the Euclidean distance version of the UCR Suite [5]. This is a set of techniques for performing very fast similarity computation scans. These optimizations include: a) avoiding the computation of square root on Euclidean distance, b) early abandoning of Euclidean distance calculations, and c) reordering early abandoning on normalized data[3]. We used these optimizations on all the methods that we examined.

Datasets. Experiments were conducted using both synthetic and real datasets. Synthetic data series were generated as random-walks (i.e., cumulative sums) of steps that follow a Gaussian distribution (0,1). This type of data has been extensively used in the past [124, 28, 135], and it is claimed to model the distribution of stock market prices [124].

Our four real datasets come from the domains of seismology, astronomy, neuroscience and image processing. The seismic dataset, *Seismic*, was obtained from the IRIS Seismic Data Access archive [140]. It contains seismic instrument recording from thousands of stations worldwide and consists of 100 million data series of size 256. The astronomy dataset, *Astro*, represents celestial objects and was obtained from [72]. The dataset consists of 100 million data series of size 256. The neuroscience dataset, *SALD*, obtained from [166] represents MRI data, including 200 million data series of size 128. The image processing dataset, *Deep1B*, retrieved from [167], contains 267 million deep network embeddings of dimension 96 extracted from the last layers of a convolutional neural network. All of our real datasets are of size 100 GB. In the rest of the paper, the size of each dataset is given in GB instead of the number of high-dimensional vectors. Overall, in our experiments, we use datasets of sizes between 25-1000GB.

Queries. All our query workloads, unless otherwise stated, include 100 query series. For synthetic datasets, we use two types of workloads: *Synth-Rand* queries are produced using the same random-walk generator (with a different seed[4]), while *Synth-Ctrl* queries are created by extracting vectors from the input data set and adding progressively larger

[3] Early abandoning of Z-normalization is not used since all datasets were normalized in advance.
[4] All seeds can be found in [93].

amounts of noise, in order to control the difficulty of each query (more difficult queries tend to be less similar to their nearest neighbor [149]). For the real datasets, query workloads are also generated by adding progressively larger amounts of noise to vectors extracted from the raw data, and we name them with the suffix -*Ctrl*. For the Deep1B dataset, we additionally include a real workload that came with the original dataset; we refer to it as *Deep-Orig*.

Scenarios. The experimental framework consists of three scenarios: parametrization, evaluation and comparison. In parametrization (§5.5.1), the optimal parameters for each method are identified. In evaluation (§3.4.2), the scalability and search efficiency for each method is evaluated under varying dataset sizes and data series lengths. Finally, in comparison (§3.4.3), methods are compared together according to the following criteria: a) scalability and search efficiency on more complex query workloads and more varied and larger datasets, b) memory and disk footprint, c) pruning ratio, and d) tightness of the lower bound.

Measures The measures we use are the following.

1. For scalability and search efficiency, we use two measures: *wall clock time* and the *number of random disk accesses*. *Wall clock time* is used to measure input, output and total elapsed times. Then CPU time is calculated as the difference between the total time and I/O time. The *number of random disk accesses* is measured for indexes. One random disk access corresponds to one leaf access for all indexes, except for the skip-sequential access method ADS+, for which one random disk access corresponds to one skip. As will be evident in the results, our measure of random disk accesses provides a good insight into the actual performance of indexes, even though we do not account for details such as caching, the number of disk pages occupied by a leaf and the numbers of leaves in contiguous disk blocks.

2. For footprint, the measures used are: *total number of nodes*, *number of leaf nodes*, *memory size*, *disk size*, *leaf nodes fill factor* and *leaf depth*.

3. We also consider the pruning ratio P, which has been widely used in the data series literature [154, 42, 25, 134, 24] as an implementation-independent measure to compare the effectiveness of an index. It is defined as follows:

$$P = 1 - \frac{\#\ of\ Raw\ Data\ Series\ Examined}{\#\ of\ Data\ Series\ In\ Dataset}$$

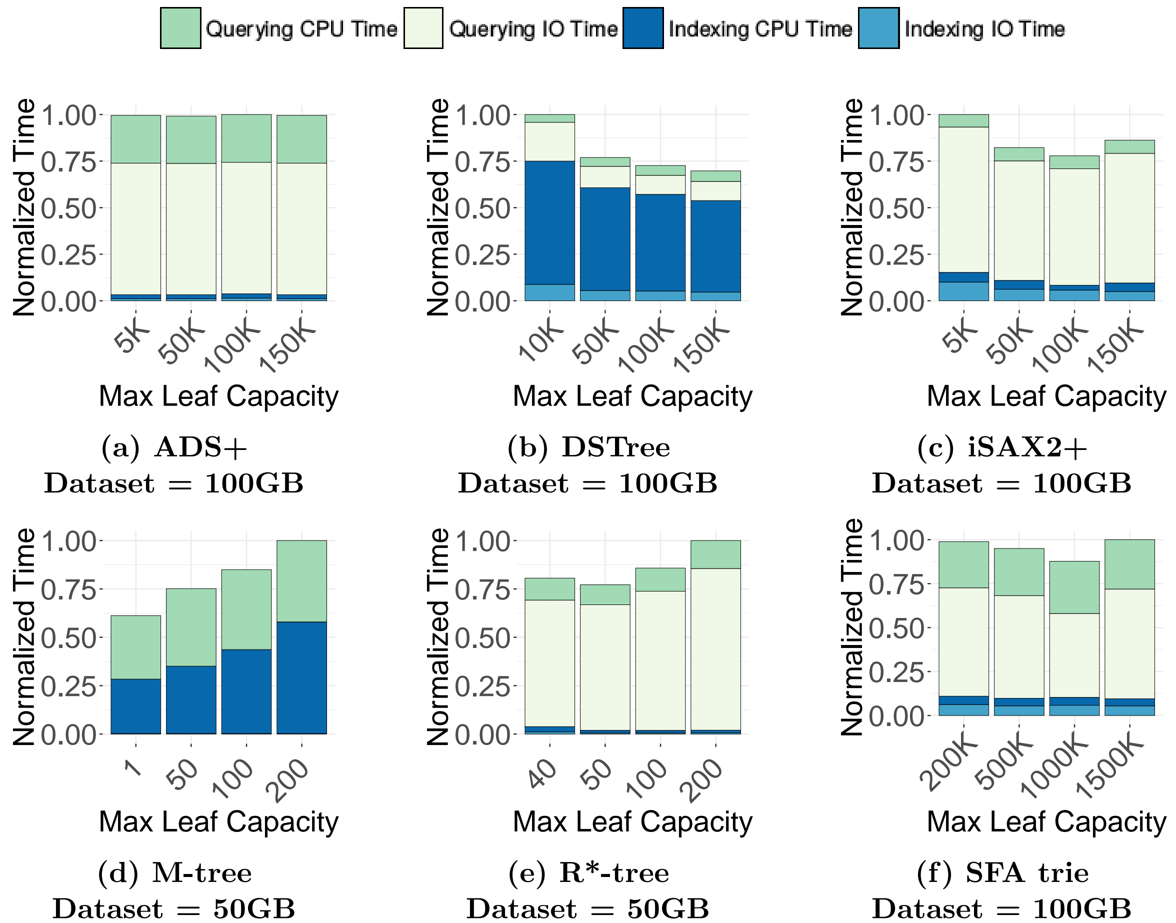
Pruning ratio is a good indicator of the number of sequential I/Os incurred. However, since relevant data series are usually spread out on disk, it should be considered along with the number of random disk accesses (seeks) performed.

4. The *tightness of the lower bound*, TLB has been used in the literature as an implementation independent measure in various different forms [146, 42, 132]. In this work we use the following version of the TLB measure that better captures the performance of indexes:

$$TLB = \frac{Lower\ Bounding\ Distance(Q\prime, N)}{Average\ True\ Distance(Q, N)}$$

Where Q is the query, $Q\prime$ is the representation of Q using the segmentation of a given leaf node N, and the average true distance between the query Q and node N is the average Euclidean distance between Q and all data series in N. We report the average over all leaf nodes for all 100 queries.

Procedure. Unless otherwise stated, experiments refer to answering 100 exact queries. Experiments with query workloads of 10,000 queries report extrapolated values. The extrapolation consists of discarding the best and worst five queries (of the original 100) in terms of total execution time, and multiplying the average of the 90 remaining queries by 10,000. Experiments involving an indexing method include a first step of building the index (or re-organizing the data as in the case of Stepwise). Caches are fully cleared before each experiment. During each experiment, the caches are warm, i.e., not cleared between indexing/preprocessing and query answering, nor after each query.



(a) ADS+ Dataset = 100GB

(b) DSTree Dataset = 100GB

(c) iSAX2+ Dataset = 100GB

(d) M-tree Dataset = 50GB

(e) R*-tree Dataset = 50GB

(f) SFA trie Dataset = 100GB

Figure 3.3: Leaf size parametrization

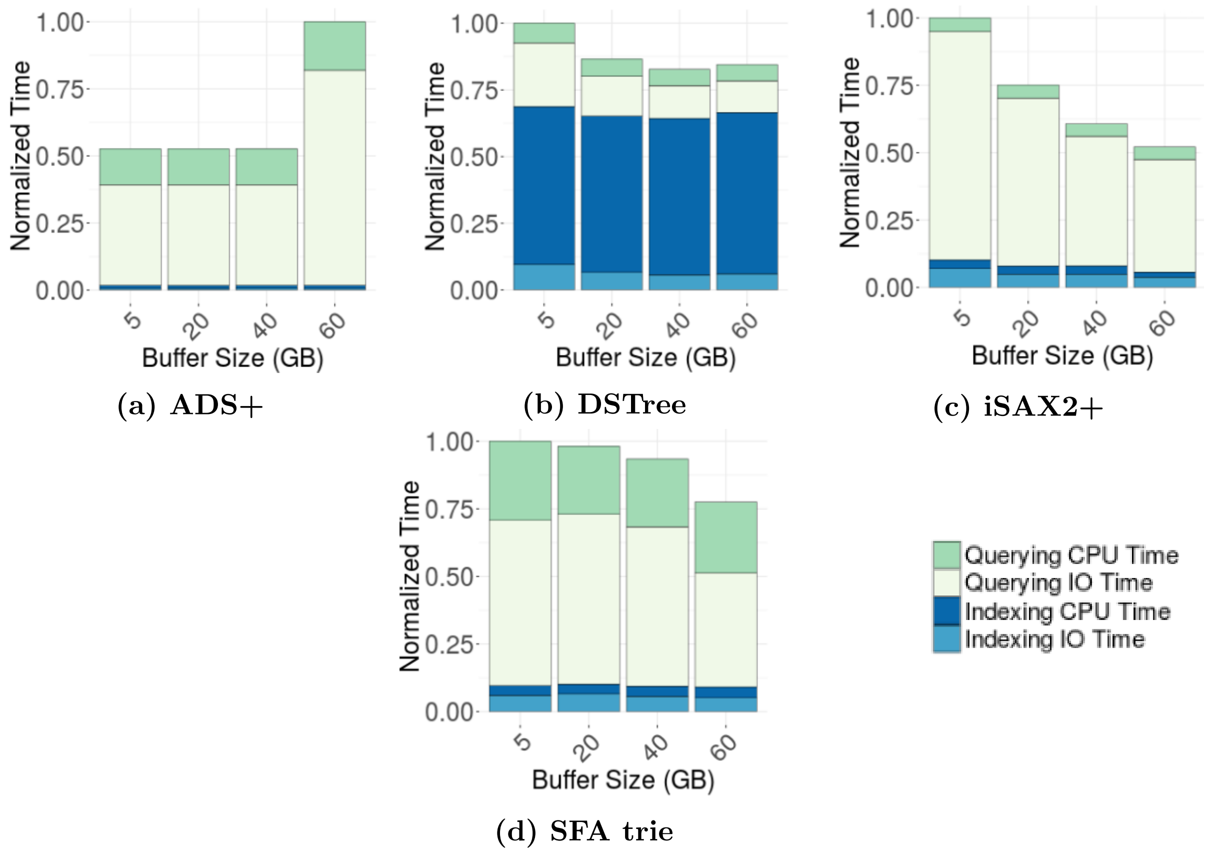


Figure 3.4: Buffer size parametrization

3.4 Results

3.4.1 Parametrization

We start our experimentation by fine tuning each method. Methods that do not support parameters are ran with their default values. The methods that support parameters are ADS+, DSTree, iSAX2+, M-tree, R*-tree and SFA trie. Since none of the methods supports auto-tuning, we perform this step manually on a synthetic dataset of 100GB with data series of length 256. Ideally, parametrization would be performed exhaustively for each scenario used in the experimental study. However, this will require an excessive amount of resources and time as a scenario is defined by 6 inputs: an algorithm, a dataset, a dataset size, a data series length, a query workload (controlled or random) and a machine (HDD or SSD). For instance, parameter tuning on the HDD machine for the random synthetic workload of the 100GB dataset of length 256 requires 30 hours. Trying all possible different combinations of parameters for the synthetic datasets alone for one algorithm would require running 7680 testing iterations. Developing auto-tuning mechanisms for these methods is an open problem which we would like to explore in future work.

The most critical parameter for these methods is the leaf threshold, i.e., the maximum number of vectors that an index leaf can hold. We thus vary the leaf size and study the tradeoffs of index construction and query answering for each method. Figure 3.3 reports indexing and querying execution times for each method, normalized by the largest total cost. The ratio is broken down into CPU and I/O times. Figure 3.3a shows that the performance of ADS+ is the same across leaf sizes. The leaf size affects indexing time, but not query answering. This is not visible in the figure, because index construction time is minimal compared to query answering time. This behavior is expected, since ADS+ is an adaptive index, which during querying splits the nodes until a minimal leaf size is reached. For M-tree, larger leaves cause both indexing and querying times to deteriorate.

For all other methods, increasing the leaf size improves indexing time (because trees are smaller) and querying time (because several series are read together), but once the leaf size goes beyond the optimal leaf size, querying slows down (because some series are unnecessarily read and processed). For DSTree, the experiments execution logs indicate that querying is faster with the 100K leaf size. The optimal leaf size for iSAX2+ is also 100K, for SFA is 1M, and for M-tree and R*-tree are 1 and 50, respectively.

SFA takes two other parameters: the alphabet size and the binning method. We ran experiments with both equi-depth and equi-width binning, and alphabet sizes from 8 (default value), to 256 (default alphabet size of iSAX2+ and ADS+). Alphabet size 8 and equi-depth binning provided the best performance and were thus used for subsequent experiments.

Some of the evaluated methods also use internal buffers to manage raw data that do not fit in memory during index building and query processing. We ran experiments varying these buffer sizes from 5GB to 60GB (Figure 3.4). The maximum was set to 60GB (recall that total RAM was 75GB). All methods benefit from a larger buffer size except ADS+. This is because a smaller buffer size allows the OS to use extra memory for file caching during query processing, since ADS+ accesses the raw data file directly.

3.4.2 Evaluation of Individual Methods

We now evaluate the indexing and search efficiency of the methods by varying the dataset size. We used two datasets of size 25GB and 50GB that fit in memory and two datasets of size 100GB and 250GB that do not fit in memory (total RAM was 75GB), with the *Synth-Rand* query workload.

ADS+. Figure 3.5a shows that ADS+ is very efficient at index building, spending most of the cost for query answering, itself dominated by the input time. The reason is that ADS+ performs skip sequential accesses on the raw data file, performing a skip almost every time a data series is pruned.

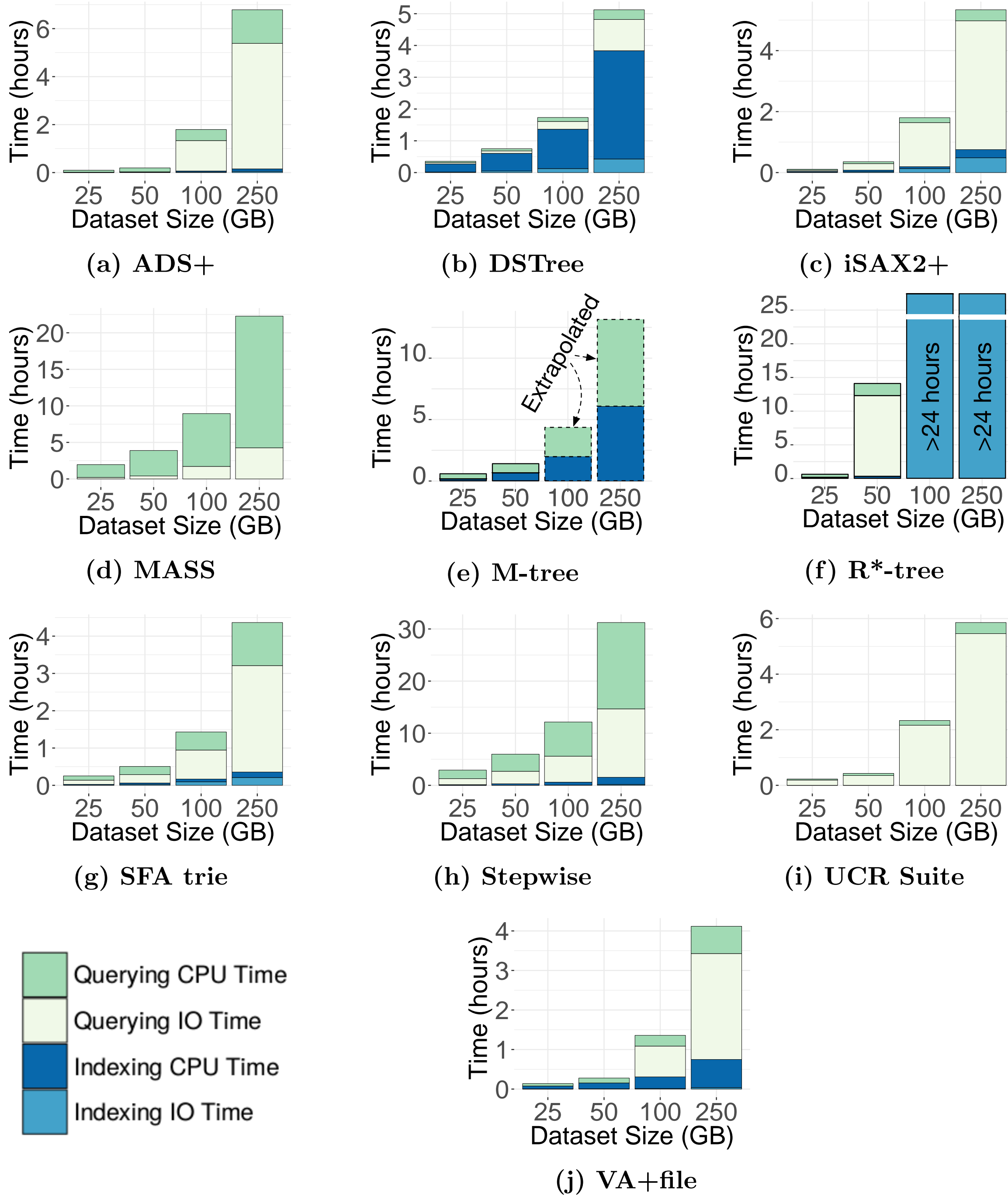
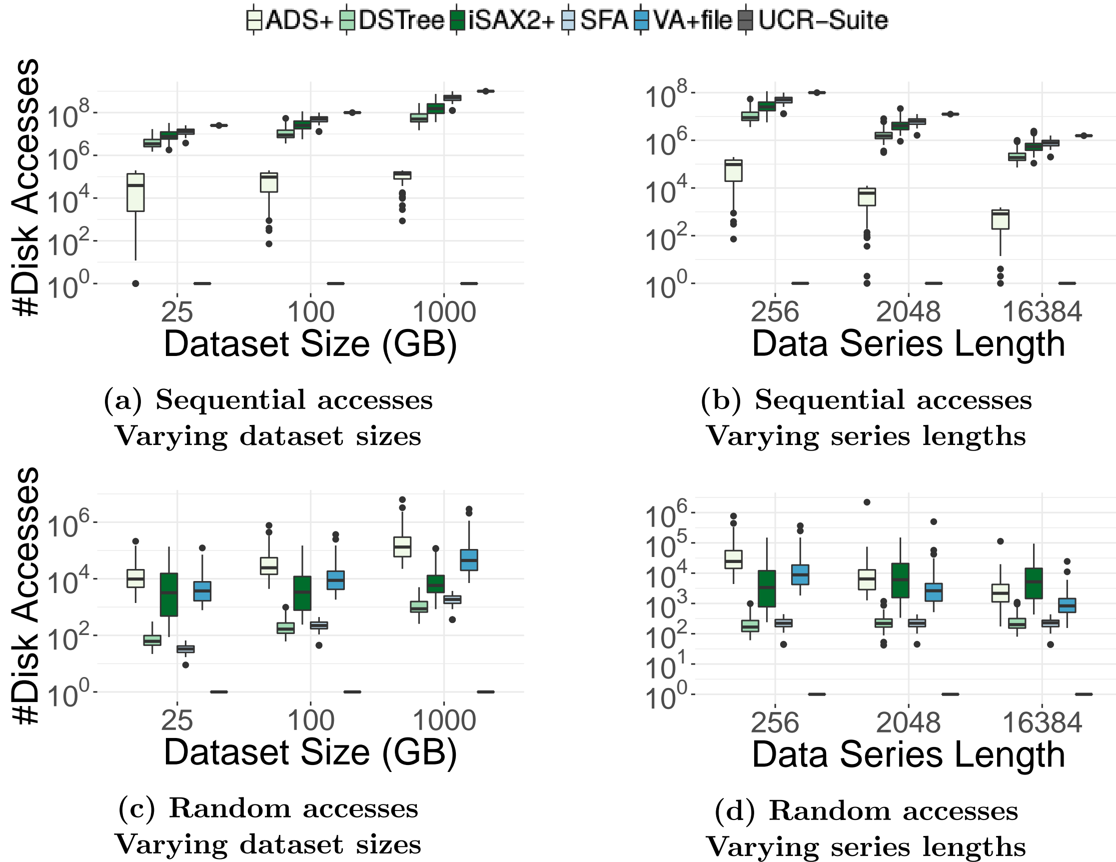


Figure 3.5: Scalability with increasing dataset sizes

DSTree. In contrast, DSTree answers queries very fast whereas index building is costly (Figure 3.5b). DSTree's cost for index building is mostly CPU, thus, offering great opportunities for parallelization.



(a) Sequential accesses Varying dataset sizes

(b) Sequential accesses Varying series lengths

(c) Random accesses Varying dataset sizes

(d) Random accesses Varying series lengths

Figure 3.6: Number of disk accesses

iSAX2+. Figure 3.5c summarizes the results for iSAX2+, which is slower to build the index compared to ADS+, but faster compared to DSTree. Query answering is faster than ADS+ and slower than the DSTree.

MASS. Figure 3.5d reports the results for MASS, which has been designed mainly for subsequence matching queries, but we adapted it for whole matching. The very high CPU cost is due to the large number of operations involved in calculating Fourier transforms and the dot product cost.

M-tree. For the M-tree, we were only able to run experiments with in-memory datasets, because the only implementation we could use is a main memory index. The disk-aware implementations did not scale beyond 1GB. Figure 3.5e shows the M-tree experimental results for the 25GB and 50GB datasets, and the (optimistic) extrapolated results for the 100GB and 250GB datasets. Note that going from 25GB to 50GB, the M-tree

performance deteriorates by a factor of 3, even though both datasets fit in memory. (The M-tree experiments for the 100GB and 250GB datasets were not able to terminate, so we report extrapolated values in the graph, by multiplying the 50GB numbers by 3 and 9, respectively, which is an optimistic estimation.) These results indicate that M-tree cannot scale to large dataset sizes.

R*-tree. Figure 3.5f shows the results for the R*-tree. Its performance deteriorates rapidly as dataset sizes increase. Even using the best implementation among the ones we tried, when the dataset reaches half the available memory, swapping causes performance to degrade. Experiments on the 100GB and 250GB datasets were stopped after 24 hours.

SFA Trie. Figure 3.5g reports the cost of index building and query processing for SFA. We observe that query processing dominates the total cost and that query cost is mostly I/O, due to the optimal leaf size being rather large.

Stepwise. Figure 3.5h indicates the time it takes for Stepwise to build the DWHT tree and execute the workload. The total cost is high and is dominated by query answering. This is because answering one query entails filtering the data level by level and requires locating the remaining candidate data corresponding to higher resolutions through random I/O.

UCR Suite. Figure 3.5i shows the time it takes for the UCR-Suite to execute the workload. Its cost is naturally dominated by input time, being a sequential scan algorithm.

VA+file. We observe in Figure 3.5j that VA+file is efficient at index building, spending most of the cost for query answering. The indexing and querying costs are dominated by CPU and input time, respectively. The CPU cost is due to the time spent for determining the bit allocation and decision intervals for each dimension; the input time is incurred when accessing the non-pruned raw vectors.

Summary. Overall, Figure 3.5 shows that it takes Stepwise, MASS, the M-tree and the R*-tree over 12 hours to complete the workload for the 250GB dataset, whereas the other methods need less than 7 hours. Therefore, in the subsequent experiments, we will only

include ADS+, DSTree, iSAX2+, SFA, the UCR suite and the VA+file.

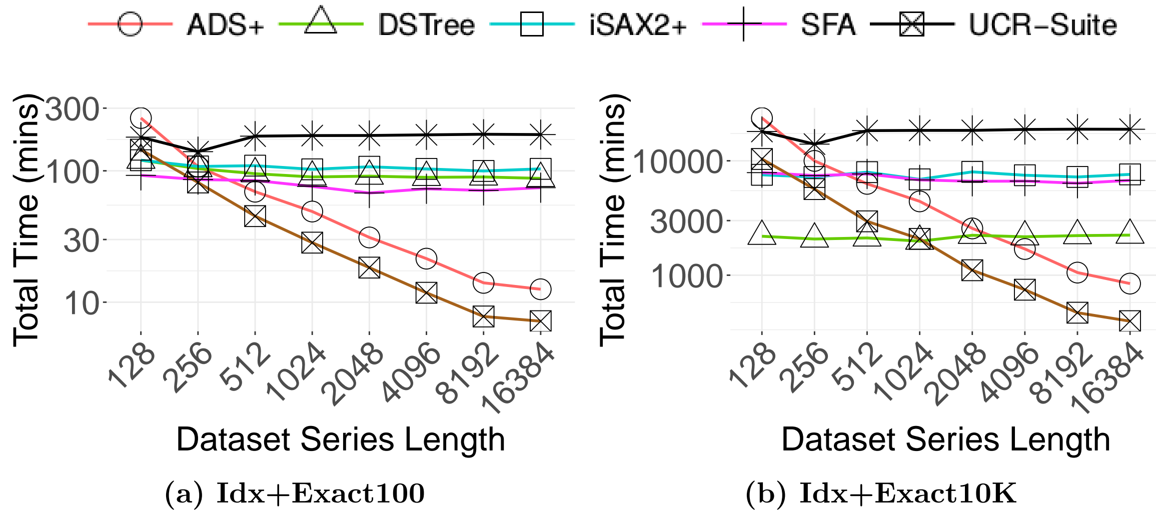
3.4.3 Comparison of the Best Methods

In the following experiments, we use the best methods as identified above, and compare them in more detail.

Disk Accesses vs Dataset Size/Sequence Length. Figure 3.6 shows the number of sequential and random disk accesses incurred by the 100 exact queries of the *Synth-Rand* workload for increasing dataset sizes and increasing lengths. When the dataset size is varied, the length of the data series is kept constant at 256, whereas the dataset size is kept at 100GB when the length is varied. We can observe that the VA+file and ADS+ perform the smallest number of sequential disk accesses across dataset sizes and data series lengths, with the VA+ performing virtually none. As expected, the UCR-Suite performs the largest number of sequential accesses regardless of the length of the series, or the size of the dataset. This number is also steady across queries, thus its boxplot is represented by a flat line. There is not a significant difference between the number of sequential operations needed by the DSTree, SFA or iSAX2+ (DSTree does the least, and SFA the most). SFA requires more sequential accesses, because its optimal leaf size is 1M, as opposed to 100K for DSTree and iSAX2+.

As far as random I/O for different dataset sizes is concerned, ADS+ performs the largest number of random accesses, followed by the VA+file. The DSTree and SFA incur almost the same number of operations. However, the DSTree has a good balance between the number of random and sequential I/O operations. It is interesting to point out that as the dataset size increases, the number of random operations for iSAX2+ becomes less skewed across queries. This is because of the fixed split-point nature of iSAX2+ that causes it to better distribute data among leaves when the dataset is large: in small dataset sizes, many leaves can contain very few series.

When the dataset size is set to 100GB and the data series length is increased, we can



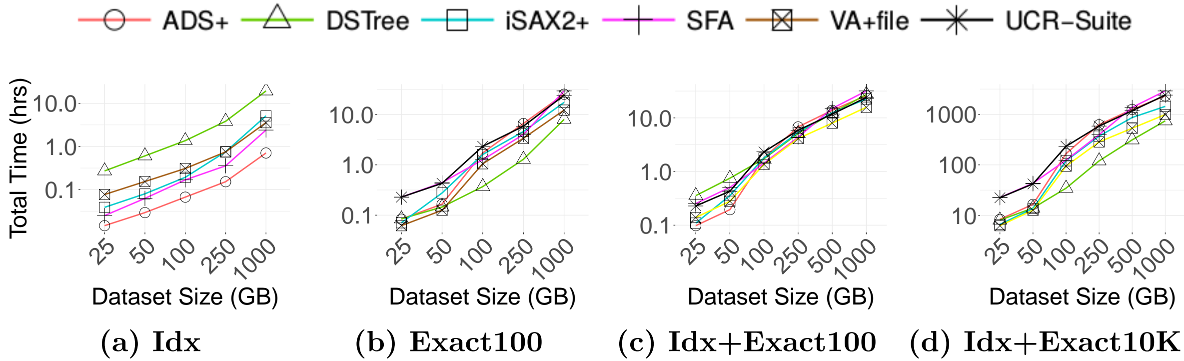
(a) Idx+Exact100 **(b) Idx+Exact10K**

Figure 3.7: Scalability with increasing lengths

observe a dramatic decrease of the number of random operations incurred by ADS+ and VA+file. The reason is that both methods use a skip-sequential algorithm, so even if the pruning ratio stays the same, when the data series is long, the algorithm skips larger blocks of data, thus the number of skips decreases. The random I/Os across lengths for the other methods remain quite steady, with SFA and DSTree performing the least.

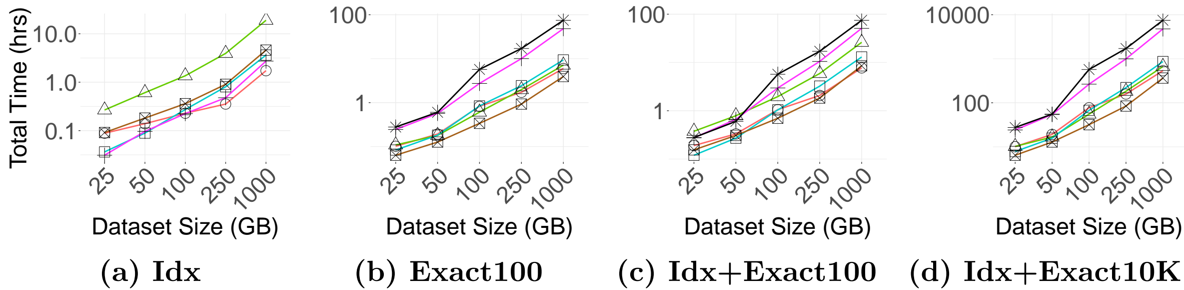
Scalability/Search Efficiency vs Sequence Length. Figure 3.7 depicts the performance of the different methods with increasing data series lengths. In order to factor out other parameters, we fix the dataset size to 100GB, and the dimensionality of the methods that use summarizations to 16, for all data series lengths. We observe that the indexing and querying costs for ADS+ and VA+file plummet as the data series length increases, whereas the cost of the other methods remains relatively steady across all lengths. This is because with increasing lengths, both algorithms perform larger sequential reads on the raw data file and fewer, contiguous skips. VA+file performs better than ADS+ since it incurs less random and almost no sequential I/Os (Figure 3.6).

Scalability/Search Efficiency vs Dataset Size - HDD. Figure 3.8 compares the scalability and search efficiency of the best methods on the HDD platform for the *Synth-Rand* workload on synthetic datasets ranging from 25GB to 1TB. There are 4 scenarios:



(a) Idx (b) Exact100 (c) Idx+Exact100 (d) Idx+Exact10K

Figure 3.8: Scalability comparison (HDD)



(a) Idx (b) Exact100 (c) Idx+Exact100 (d) Idx+Exact10K

Figure 3.9: Scalability comparison (SSD)

indexing (Idx), answering 100 exact queries (Exact100), indexing and answering 100 exact queries (Idx+Exact100), and indexing and answering 10,000 queries (Idx+Exact10K). Times are shown in log scale to reveal the performance on smaller datasets.

Figure 3.8a indicates only the indexing times. ADS+ outperforms all other methods and is an order of magnitude faster than the slowest, DSTree. Figure 3.8b shows the times for running 100 exact queries. We observe two trends in this plot. For in-memory datasets, VA+file surpasses the other methods. For the larger datasets, the DSTree is a clear winner, followed by VA+file, while the performance of the other methods converge to that of sequential scan. Figure 3.8c refers to indexing and answering the 100 exact queries. For in-memory datasets, ADS+ shows the best performance, with iSAX2+ performing equally well on the 25GB dataset. However, for larger datasets, VA+file outperforms all other methods.

Figure 3.8d shows the time for indexing and answering 10K exact queries. The trends now change. For in-memory datasets, iSAX2+ and VA+file outperform all other meth-

ods, in particular ADS+. Both iSAX2+ and VA+file are slower than ADS+ in index building, but this high initial cost is amortized over the large query workload.

The DSTree is the best contender for large data sets that do not fit in memory, followed by VA+file and iSAX2+. The other methods perform similar to a sequential scan. The DSTree has the highest indexing cost among these methods, but once the index is built, query answering is very fast, thus being amortized for large query workloads. The strength of the DSTree is based on its sophisticated splitting policy, the upper/lower bounds used in query answering, and its parameter-free summarization algorithm.

Our results for in-memory datasets corroborate earlier studies [30] (i.e., ADS+ outperforms alternative methods), yet, we additionally bring in the picture VA+file, which is very competitive and had not been considered in earlier works. Moreover, for out-of-memory data, our results show that ADS+ is not faster than sequential scan, as was previously reported. The reason for this discrepancy in results lies with the different hardware characteristics, which can significantly affect the performance of different algorithms, both in relative, as well as in absolute terms. More specifically, the disks used in [30] had 60% of the sequential throughput of the disks used in this paper. As a result, ADS+ can be outperformed by a sequential scan of the data when the disk throughput is high and the length of the sequences is small enough, where ADS+ is forced to perform multiple disk seeks. Figures 3.6a and 3.6c clearly show that ADS+ performs the smallest number of sequential disk operations and the largest number of random disk operations across all datasets. In main-memory, SSDs, and with batched I/Os, ADS+ is expected to perform significantly better.

Scalability/Search Efficiency vs Dataset Size - SSD. In order to further study the effect of different hardware on the performance of similarity search methods, we repeated the experiments described in the last paragraph on the SSD machine. We once again tuned each index on the 100GB dataset to find the optimal leaf threshold, which this time was an order of magnitude smaller than the optimal leaf size for the HDD platform.

However, we were not able to perform experiments with our larger datasets with these smaller leaf sizes, because the maximum number of possible split points was reached before indexing the entire dataset. Although small leaf sizes can improve performance on smaller datasets, they cannot be used in practice, since the index itself cannot be constructed. Therefore, we iteratively increased the leaf sizes, and picked the ones that worked for all datasets in our experiments: these leaf sizes proved to be the same as the ones for the HDD platform. We note that the SFA trie was particularly sensitive to parametrization.

There are two main observations on these results (see Figure 3.9). The first is that VA+file and ADS+ are now the best performers on most scenarios. The only exceptions are iSAX2+ surpassing ADS+ on the 25GB workload, and iSAX2+/SFA being faster in indexing the in-memory datasets. As discussed earlier, the bottleneck of ADS+ and VA+file is random I/O, so the fast performance of the SSD machine on random I/O explains why they both win over the other methods. ADS+ is faster than VA+file at indexing, while the opposite is true for query answering. The indexing cost of VA+file is amortized in the 10K workload. The second observation is that UCR-Suite performs poorly, due to the low disk throughput of the SSD server.

Memory/Disk Footprint vs Dataset Size. In this set of experiments, we compare the disk and memory footprints of all methods. Figure 3.10a shows that SAX-based indexes have the largest number of nodes. SFA has a very low number of nodes, because the leaf size we use is 1,000,000 (refer to Figure 3.3), whereas the leaf sizes for DSTree and iSAX2+ are both 100,000. The ADS+ index is indifferent to leaf size so we set its initial value to 100,000. For all methods, most nodes are leaves, as shown in Figure 3.10b. Note that ADS+ and iSAX2+ have the same tree structure with en equal number of nodes, since the leaf size is the same.

As shown in Figures 3.10c and 3.10d, the size of the indexes in memory and on disk follows the same trend as the number of nodes. Although ADS+ and iSAX2+ have the

Table 3.2: Controlled workloads experimental results summary (sequential scan algorithm is highlighted)

		Scenarios					
	Dataset	Idx	Exact 100	Idx+ Exact 100	Idx+ Exact 10K	Exact Easy-20	Exact Hard-20
HDD	Small	A	D	S	D	D	D
HDD	Large	A	D	S	D	D	D
HDD	Astro	A	U	U	V	V	U
HDD	Deep1B	A	U	U	U	D	U
HDD	SALD	A	D	I	D	D	D
HDD	Seismic	A	D	S	D	D	U
SSD	Small	S	D	I	D	I	D
SSD	Large	S	D	I	D	I	D
SSD	Astro	I	V	V	V	V	V
SSD	Deep1B	S	I	I	V	I	U
SSD	SALD	S	I	I	I	I	V
SSD	Seismic	A	V	V	V	D	V

A: ADS, **D:** DSTree, **I:** iSAX2+
S: SFA, **U:** UCR-Suite, **V:** VA+file

same tree shape, some of the data types and structures they use are not the same, thus the different sizes in memory. For the VA+file, we only report the size of the approximation file on disk, since it does not build an auxiliary tree structure.

We use two measures to compare the overall structure of the indexes. The first is the leaf nodes fill factor, which measures the percentage of the leaf that is full, and gives a good indication of whether the index distributes evenly the data among leaves. The second measure is the depth of the leaves, which can help evaluate how balanced an index is. While none of the best performing index trees is truly height-balanced, some are better balanced in practice than others. Figure 3.10e shows the leaf nodes fill factor for different dataset sizes and methods. (Note that VA+file is missing, since it has no tree; though, if we consider as leaves the pages, where it stores the data, then the fill factor of these pages is 100%.) We observe that SFA offers the least variability in the fill factor for the small datasets (as indicated by the size of the boxplot), but the median fill factor fluctuates as the data set changes. DSTree provides the highest median fill factor

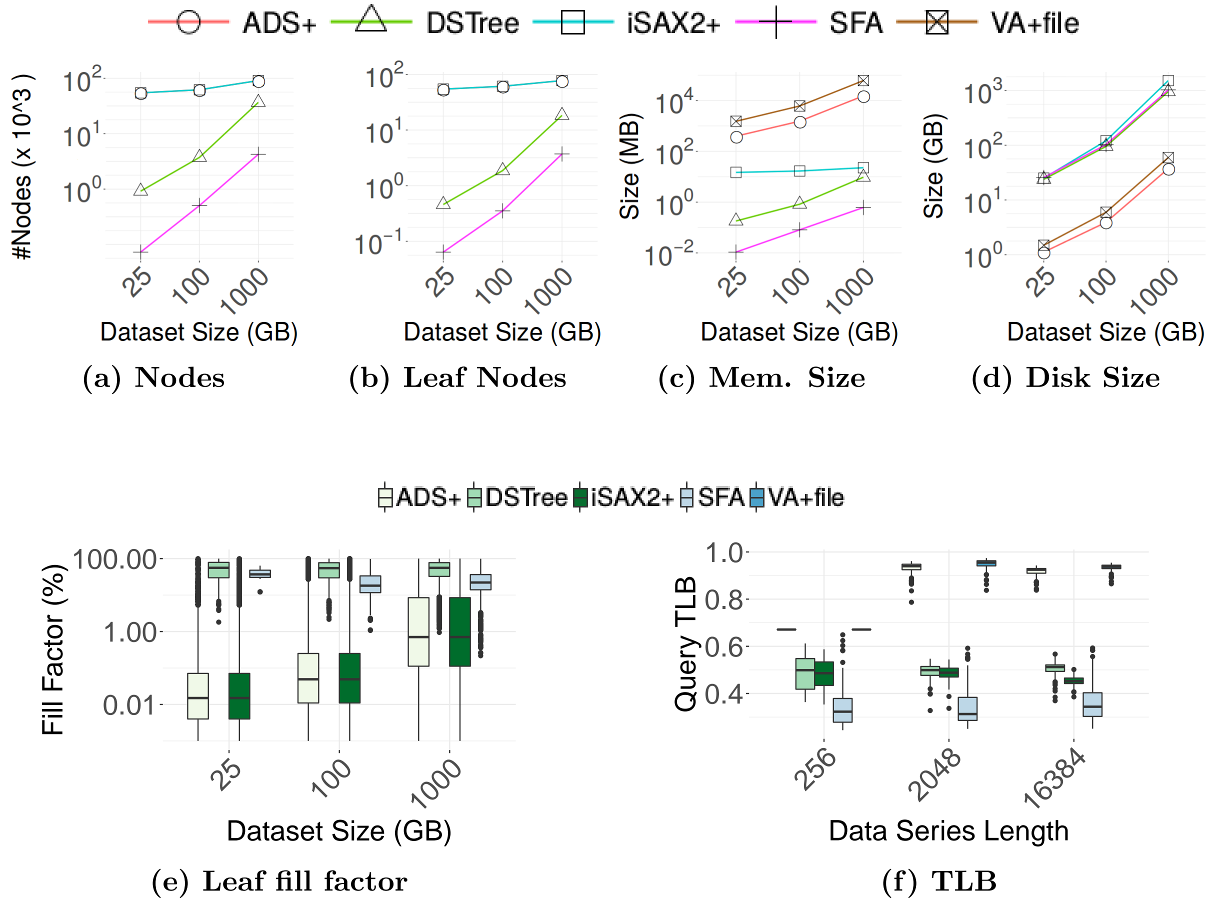


Figure 3.10: Exact methods footprint and TLB for synthetic datasets

(as indicated by the line in the middle of the boxplot), which also remains steady with increasing data set sizes. DSTree also displays the least skew and virtually no outliers, which means that this index effectively partitions the dataset and distributes the series across all its leaves. The SAX-based indexes have many outliers, with some leaves being full and others being empty. The graph showing the depth of the indexes can be found elsewhere [93].

Tightness of the Lower Bound. Figure 3.10f shows the TLB (defined in Section 4.4.2) of each method for increasing data series lengths. We observe that the TLBs of ADS+ and VA+file increase rapidly with increasing lengths, then stabilize when they reach a value close to 1. This explains why the performance of both methods improves with longer series. We also note that VA+file has a slightly tighter lower bound than ADS+,

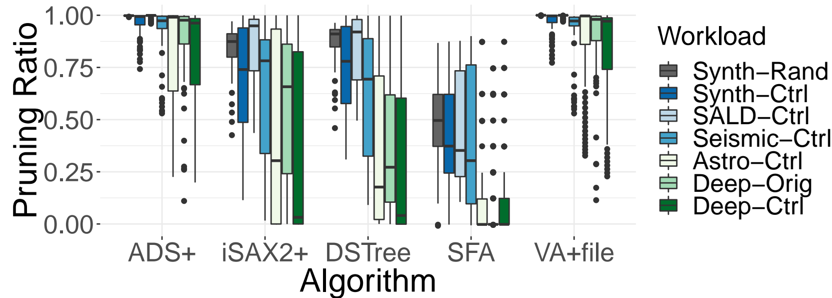


Figure 3.11: Pruning ratio (Dataset Size= 100GB, Workload = 100 Queries)

thanks to its non-uniform discretization scheme, which helps explain why VA+file incurs less random I/O than ADS+, and thus performs better. The TLB of the SFA trie is low compared to the other methods, although we used the tight lower bounding distance of SFA (which uses the DFT MBRs). We believe this is due to the optimal alphabet size of 8, which is rather small compared to the default alphabet size of 256 for the SAX-based methods. As for iSAX2+ and DSTree, the main difference in the TLB is that it becomes virtually constant as the length increases.

Pruning Ratio. We measure the pruning ratio (higher is better) for all indexes across datasets and data series lengths. For the *Synth-Rand* workload on synthetic datasets, we varied the size from 25GB to 1TB and the length from 128 to 16384. We observed that the pruning ratio remained stable for each method and that overall ADS+ and VA+file have the best pruning ratio, followed by DSTree, iSAX2+ and SFA. We also ran experiments with a real workload (*Deep-Orig*), a controlled workload on the 100GB synthetic dataset (*Synth-Ctrl*), and controlled workloads on the real datasets (*Astro-Ctrl*, *Deep-Ctrl*, *SALD-Ctrl* and *Seismic-Ctrl*). In the controlled workloads, we extract series from the dataset and add noise. Figure 3.11 summarizes these results. For lack of space, we only report the pruning ratio for the real datasets (all of size 100GB) and the 100GB synthetic dataset. The pruning ratio for *Synth-Rand* is the highest for all methods. We observe that the *Synth-Ctrl* workload is more varied than *Synth-Rand* since it contains harder queries with lower pruning ratios. The trend remains the same with ADS+ and

VA+file having the best pruning ratio overall, followed by DSTree, iSAX2+ then SFA. For real dataset workloads, ADS+ and VA+file achieve the best pruning, followed by iSAX2+, DSTree, and then SFA. The relatively low pruning ratio for the SFA is most probably due to the large leaf size of 1,000,000. Once a leaf is retrieved, SFA accesses all series in the leaf, which reduces the pruning ratio significantly. VA+file has a slightly better pruning ratio than ADS+, because it performs less random and sequential I/O, thanks to its tighter lower bound. We note that the pruning ratio alone does not predict the performance of an index. In fact, this ratio provides a good estimate of the number of sequential operations that a method will perform, but it should be considered along with other measures like the number of random disk I/Os.

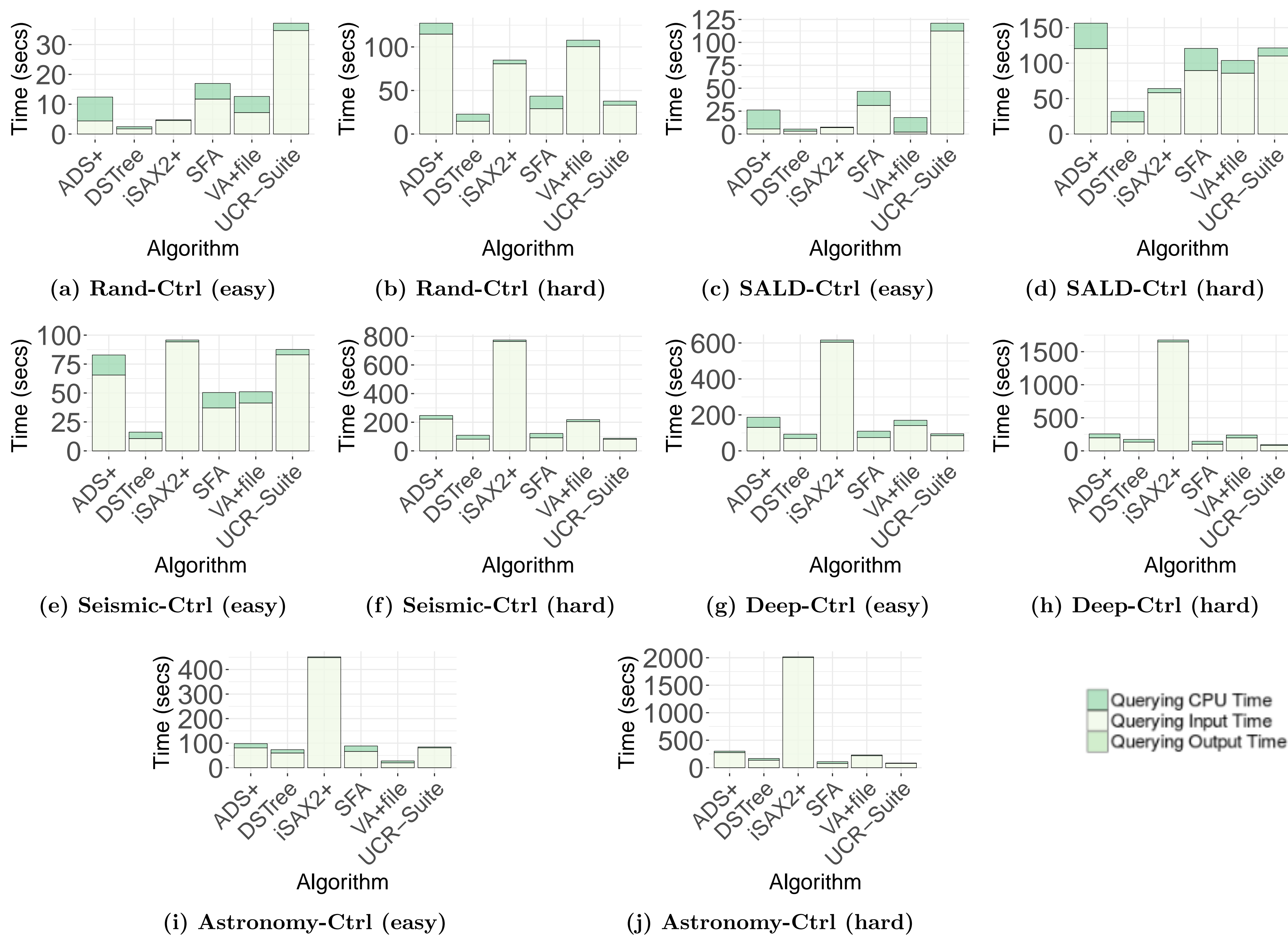
Scalability/Search Efficiency with Real Datasets. In Table 3.2, we report the name of the best method for each scenario. In addition to the four scenarios discussed earlier, we also consider two new scenarios: the average time of the 20 easiest queries (Easy-20) and the average time of the 20 hardest queries (Hard-20) of the corresponding workload. A query is considered easy, or hard, depending on its pruning ratio (computed as the average across all techniques) [149].

It is important to note that while queries are categorized as easy and hard, easy queries on one dataset may be harder than easy queries on another dataset, as the average pruning ratio for each dataset differs. This is because some datasets can be summarized more effectively than others. We averaged the results over 20 hard queries and 20 easy queries. In-memory datasets are labeled *small* and the others *large*.

We observe that UCR-Suite wins in exact query answering and on hard queries for the Astro/Deep1B scenarios. This is due to the very low pruning ratio for these workloads. DSTree is fast on easy queries and exact query answering on the SALD/Seimic scenarios. ADS+ always wins in indexing on HDD, but is sometimes surpassed by iSAX2+/SFA on SSD. Similarly to synthetic datasets, the methods behave differently on real datasets

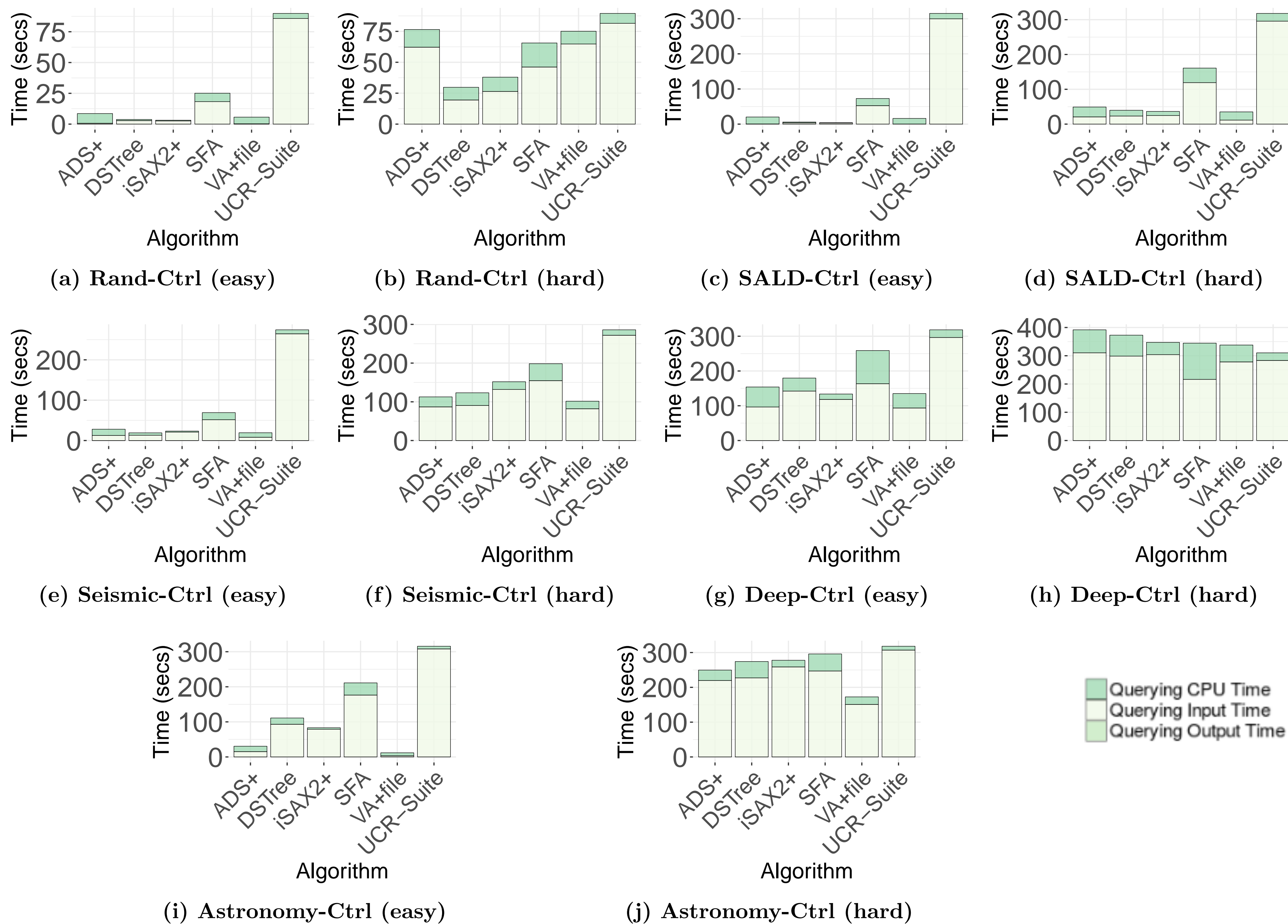
when experiments are ran on the SSD platform. VA+file and iSAX2+ have a superior performance overall. DSTree also performs well, while UCR-Suite wins only on hard queries on the Deep1B dataset.

The complete results for the controlled workloads on the both the HDD and SSD platforms are found in Figures 3.12 to 3.17.



(a) **Rand-Ctrl (easy)**
(b) **Rand-Ctrl (hard)**
(c) **SALD-Ctrl (easy)**
(d) **SALD-Ctrl (hard)**
(e) **Seismic-Ctrl (easy)**
(f) **Seismic-Ctrl (hard)**
(g) **Deep-Ctrl (easy)**
(h) **Deep-Ctrl (hard)**
(i) **Astronomy-Ctrl (easy)**
(j) **Astronomy-Ctrl (hard)**

Figure 3.12: Average time of queries with different difficulty (HDD)

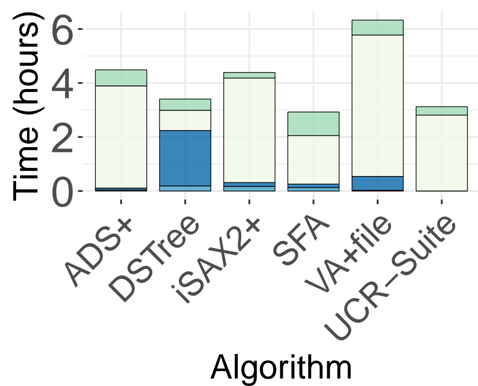


(a) **Rand-Ctrl (easy)** (b) **Rand-Ctrl (hard)** (c) **SALD-Ctrl (easy)** (d) **SALD-Ctrl (hard)**

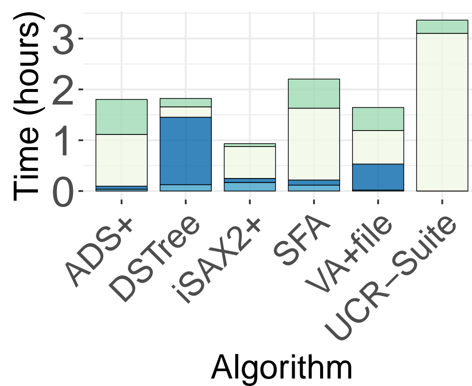
(e) **Seismic-Ctrl (easy)** (f) **Seismic-Ctrl (hard)** (g) **Deep-Ctrl (easy)** (h) **Deep-Ctrl (hard)**

(i) **Astronomy-Ctrl (easy)** (j) **Astronomy-Ctrl (hard)**

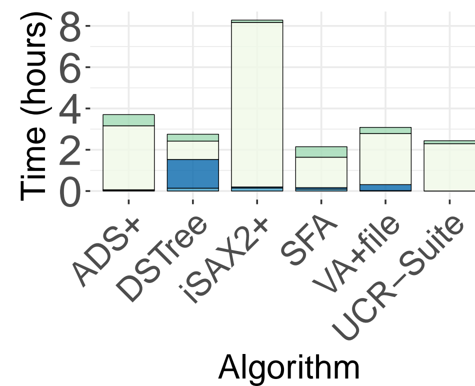
Figure 3.13: Average time of queries with different difficulty (SSD)



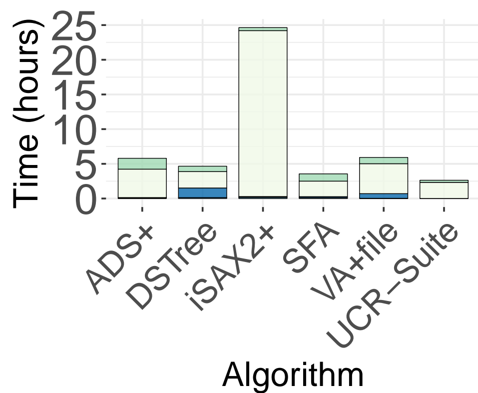
(a) **Rand-Ctrl**



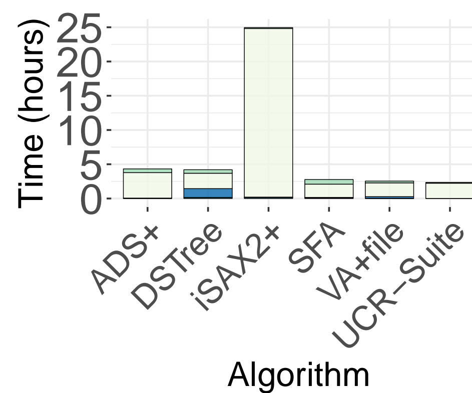
(b) **SALD-Ctrl**



(c) **Seismic-Ctrl**



(d) **Deep-Ctrl**



(e) **Astronomy-Ctrl**

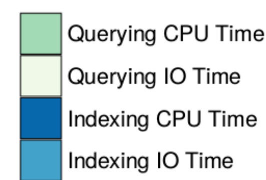
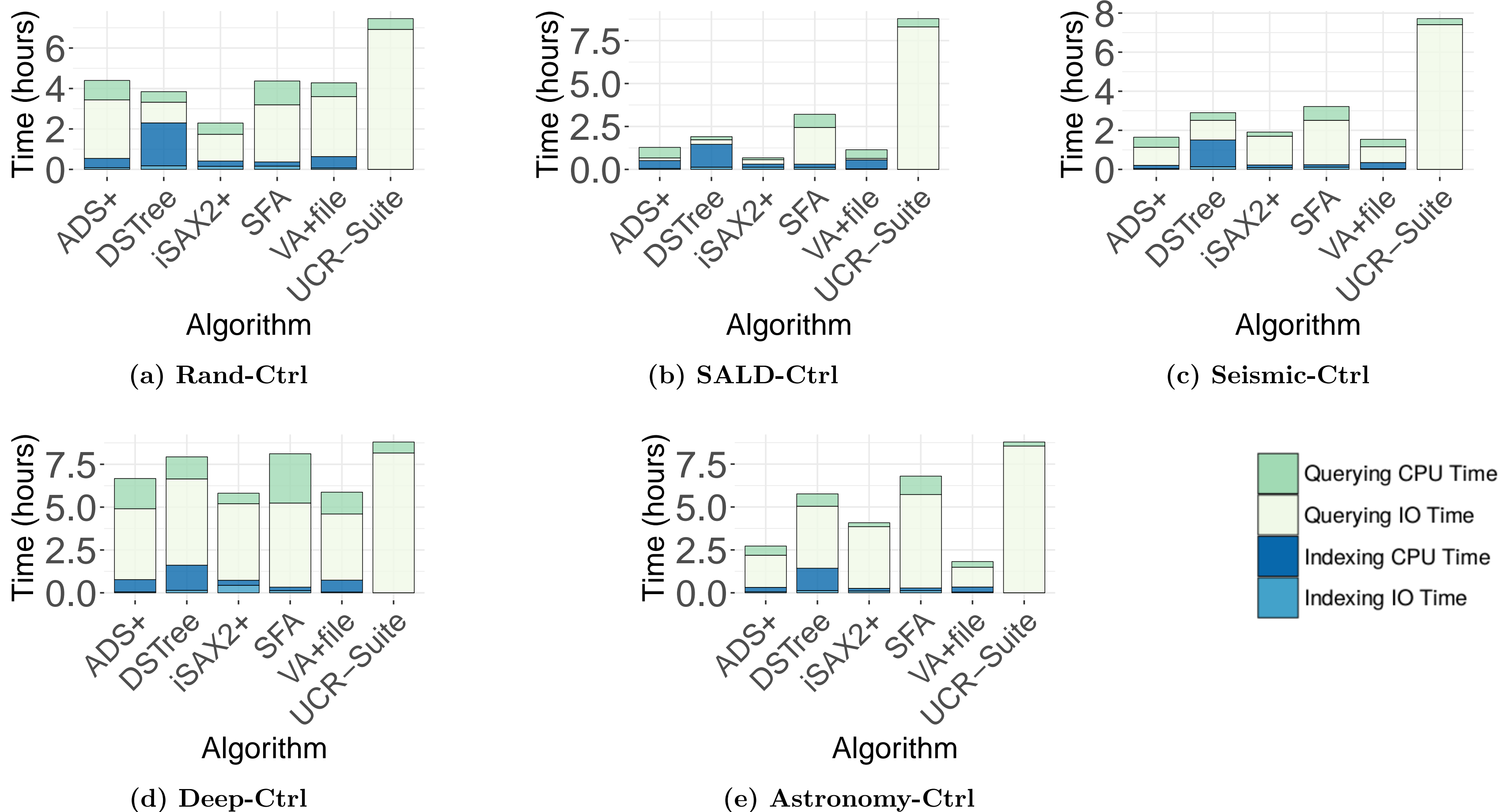


Figure 3.14: Indexing and answering 100 queries (HDD)



(a) **Rand-Ctrl**

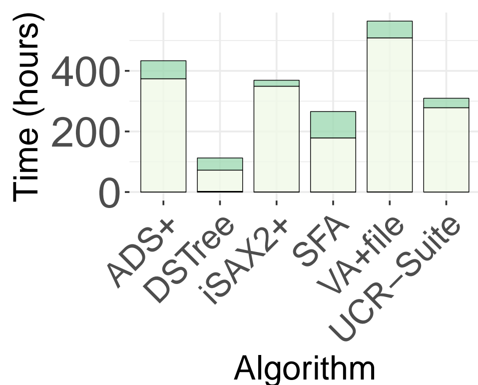
(b) **SALD-Ctrl**

(c) **Seismic-Ctrl**

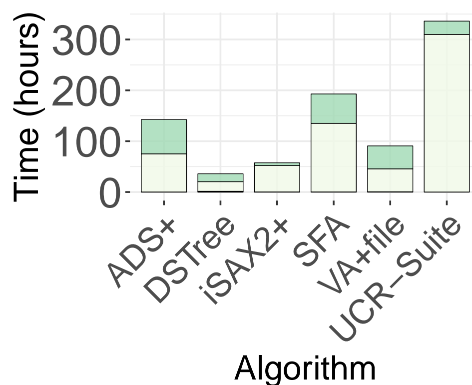
(d) **Deep-Ctrl**

(e) **Astronomy-Ctrl**

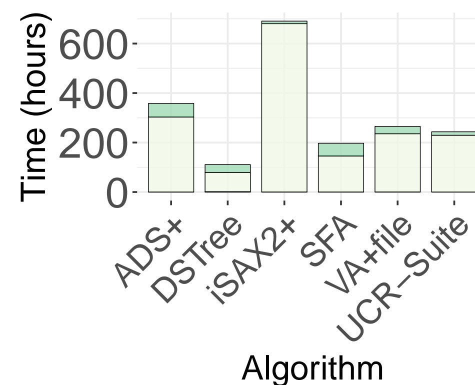
Figure 3.15: Indexing and answering 100 queries (SSD)



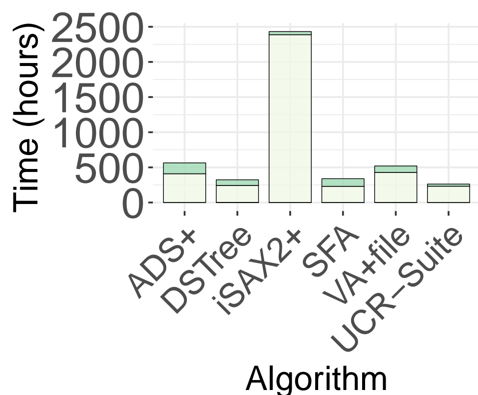
(a) **Rand-Ctrl**



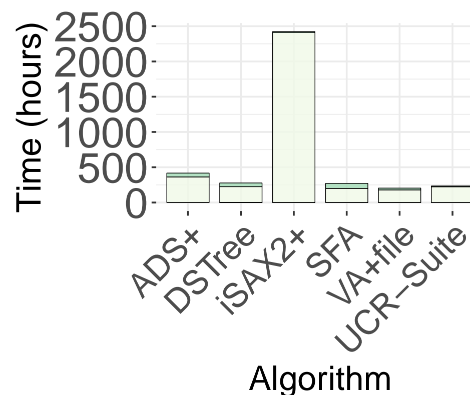
(b) **SALD-Ctrl**



(c) **Seismic-Ctrl**



(d) **Deep-Ctrl**



(e) **Astronomy-Ctrl**

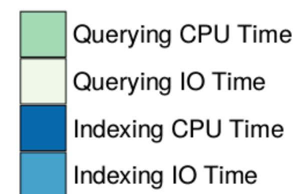
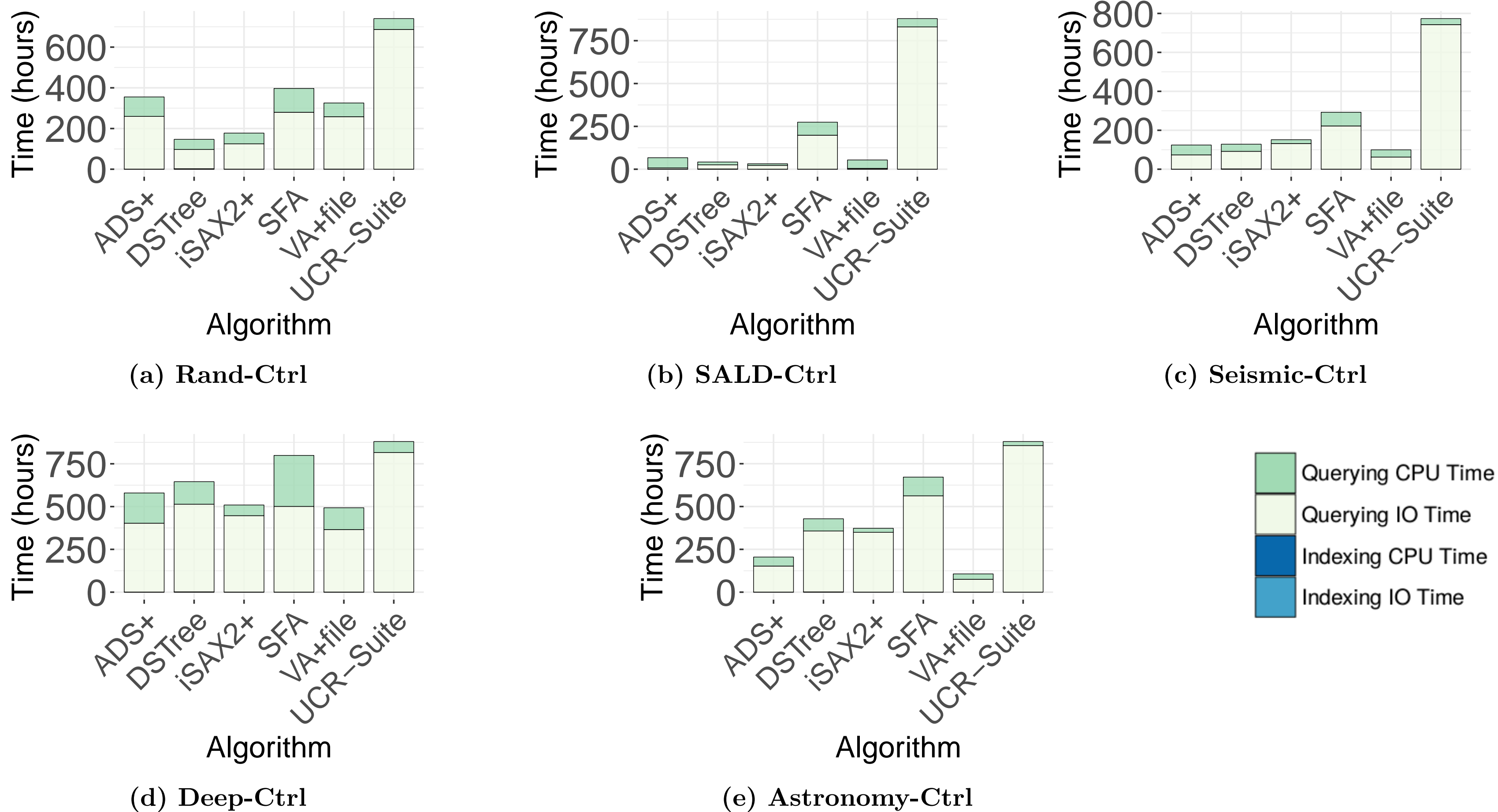


Figure 3.16: Indexing and answering 10K queries (HDD)



(a) Rand-Ctrl

(b) SALD-Ctrl

(c) Seismic-Ctrl

(d) Deep-Ctrl

(e) Astronomy-Ctrl

Figure 3.17: Indexing and answering 10K queries (SSD)

3.5 Discussion

In the data series literature, competing similarity search methods have never been compared together under a unified experimental scheme. The objective of this experimental evaluation is to consolidate previous work on data series whole-matching similarity search and prepare a solid ground for further developments in the field.

We undertook a challenging and laborious task, where we re-implemented from scratch four algorithms: iSAX2+, SFA trie, DSTree, and VA+file, and optimized memory management problems (swapping, and out-of-memory errors) in R*-tree, M-tree, and Stepwise. Choosing C/C++ provided considerable performance gains, but also required low-level memory management optimizations. We believe the effort involved was well worth it since the results of our experimental evaluation emphatically demonstrate the importance of the experimental setup on the relative performance of the various methods. To further facilitate research in the field we publicize our source code and experimental results [93]. This section summarizes the lessons learned in this study.

Unexpected Results. For some of the algorithms our experimental evaluation revealed some unexpected results.

(1) *The Stepwise method performed lower than our expectations.* This was both due to the fact that our baseline sequential scan was fully optimized for early abandoning and computation optimization, but most importantly because of *a different experimental setup.* The original implementation of Stepwise performed batched query answering. In our case we compared all methods on single query at a time workload scenario. This demonstrates the importance of *the experimental setup and workload type.*

(2) *The VA+file method performed extremely well.* Although an older method, VA+file is among the best performers overall. Our optimized implementation, which is much faster than the original version, helped unleash the best of this method; this demonstrates the importance of *the implementation framework.*

(3) *For exact queries on out-of-memory data on the HDD machine, ADS+ is under-*

performing. The reason is that ADS+ performs multiple skips while pruning at a per series level and is thus significantly affected by the hard disk's latency. In the original study [30], ADS+ was run on a machine with 60% of the hard disk throughput of the one used in the current work. The HDD setup with the 6 RAID0 disks gave a significant advantage on methods that perform sequential scans on larger blocks of data and less skips. On the SSD machine, however, the trend is reversed, and ADS+ becomes one of the best contenders overall. These observations demonstrate the importance of *the hardware setup.*

(4) *The optimal parameters of most algorithms were different than the ones presented in their respective papers.* This is because some methods were not tuned before: the iSAX2+, DSTree and SFA papers have no tuning experiments. We tuned each for varying leaf and buffer sizes (for brevity, we only report results for leaf parametrization in Figure 3.3 (for buffer tuning experiments, see [93]). For SFA, we also tuned the sample size used to identify the breakpoints, binning method (equi-depth vs. equi-width), and number of symbols for the SFA discretization. Another reason is that we studied in more detail methods that were partially tuned (e.g., ADS+ was tuned only for varying leaf size; we also varied buffer size and found that assigning most of RAM to buffering hurts performance). These findings further demonstrate the need for *careful parameter-tuning.*

(5) *The quality of the summarization, as measured by TLB and pruning, is not necessarily correlated to time performance.* An early experimental study [148] claimed that the tightness of the lower bound can be used alone to evaluate the efficiency of indexing techniques. While summarization quality is an important factor on its own, we demonstrate that it cannot alone predict the time performance of an index, even in the absence of data and implementation biases. For example, ADS+ achieves very high pruning and TLB, yet, in terms of time, it is outperformed by other methods in some scenarios. It is of crucial importance to consider summarization quality alongside the properties of the index structure and the hardware platform.

Speed-up Opportunities. Through our analysis, we identified multiple factors that affect the performance of examined methods. In turn, these factors reveal opportunities and point to directions for performance improvements.

(1) *Stepwise* offers many such avenues. Its storage scheme could be optimized to reduce the number of random I/O during query answering, and its query answering algorithm would benefit a lot from parallelization and modern hardware optimizations (i.e., through multi-core and SIMD parallelism), as 50%-98% of total time is CPU.

(2) *DSTree* is very fast at query answering, but rather slow at index building. Nevertheless, a large percentage of this time (85-90%) is CPU. Therefore, also the indexing performance of DSTree can be improved by exploiting modern hardware. Moreover, bulk loading during indexing, and buffering during querying, would also make it even faster.

(3) A similar observation holds for *VA+file* and *MASS*. Even though MASS is not designed for whole-matching data series similarity search, its performance can be significantly enhanced with parallelism and modern hardware exploitation, since 90% of its execution time is CPU cost. Similarly, the indexing cost of VA+file can be further improved.

(4) Finally, we obtained a better understanding of the *ADS+* algorithm. Apart from being very fast in index building, our results showed that it also has a leading performance for whole-matching similarity search for *long* data series. We also discovered that the main bottleneck for ADS+ are the multiple skips performed during query answering. Its effects could be masked by controlling the size of the data segments skipped (i.e., skipping/reading large continuous blocks), and through asynchronous I/O. Moreover, because of its very good pruning power (that leads to an increased number of skips), we expect ADS+ to work well whenever random access is cheap, e.g., with SSDs and main-memory systems.

Data-adaptive Partitioning. While the SFA trie and iSAX-based index building algorithms are much faster than the DSTree index building algorithm, their performance

during query answering is much worse than that of DSTree. DSTree spends more time during indexing, intelligently adapting its leaf summarizations when split operations are performed. This leads to better data clustering and as a result faster query execution. On the contrary, both iSAX and SFA have fixed maximum resolutions, and iSAX indexes can only perform splits on predefined split-points. Even though iSAX summarizations at full resolution offer excellent pruning power (see ADS+ in Figure 3.11), grouping them using fixed split-points in an iSAX-based index does not allow for effective clustering (see Figure 3.10e). This is both an advantage (indexing is extremely fast), but also a drawback as it does not allow clustering to adapt to the dataset distribution.

Access-Path Selection. Finally, our results demonstrate that the pruning ratio, along with the ability of an index to cluster together similar data series in large contiguous blocks of data, is crucial for its performance. Moreover, our results confirm the intuitive result that the smaller the pruning ratio, the higher the probability that a sequential scan will perform better than an index, as can be observed for the hard queries in Table 3.2. This is because it will avoid costly random accesses patterns on a large part of the dataset. However, the decision between a scan or an index, and more specifically, the choice of an index, is not trivial, but is based on a combination of factors: (a) the effectiveness of the summarization used by the index (which can be estimated by the pruning ratio); (b) the ability of the index to cluster together similar data series (which determines the access pattern); and (c) the hardware characteristics (which dictate the data access latencies). This context gives rise to interesting optimization problems, which have never before been studied in the domain of data series similarity search.

Recommendations. Figure 4.10 presents a decision matrix that reports the best approach to use for problems with different data series characteristics, given a specific hardware setup (i.e., HDD) and query workload (i.e., Indexing + 10K synthetic queries). In general though, choosing the best approach to answer a similarity query on massive data series is an optimization problem, and needs to be studied in depth.

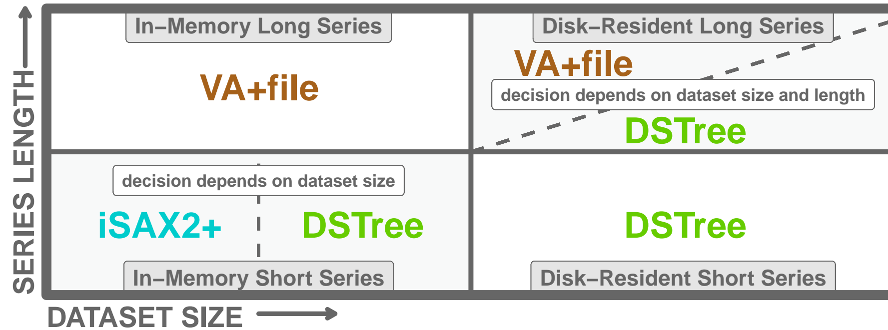


Figure 3.18: Recommendations (Indexing and answering 10K queries on HDD)

3.6 Conclusions

In this chapter, we presented the results of the most thorough experimental comparison in the literature of the state-of-the-art exact high-dimensional similarity search methods, which had never before been compared at equal footing to one another. We examined which approaches are good candidates for parallelization and modern hardware optimizations after assessing their strengths and weaknesses. We also pin-pointed the approaches which better withstand the curse of dimensionality, those which scale better with large datasets and workloads and identified the DSTree, iSAX2+ and the VA+file to be the best choices overall. We introduced for the first time in an experimental evaluation, query workloads that differentiate between easy and hard queries, which enable to stress-test a wide variety of high-dimensional similarity search methods. Our results paint a clear picture of the strengths and weaknesses of the various approaches, and indicate promising research directions.

Chapter 4

Approximate Similarity Search

Data series are a special type of high-dimensional data present in numerous domains, where similarity search is a key operation that has been extensively studied in the data series literature, leading to the development of efficient exact indexing methods. In parallel, the high-dimensional community has studied approximate similarity search techniques.

In this chapter[1], we propose a new class of efficient approximate similarity search techniques with theoretical guarantees based on modifications to exact data series indexes, and conduct a thorough experimental evaluation to compare these techniques to state-of-the-art methods under a unified framework. Although data series differ from generic high-dimensional vectors (series usually exhibit correlation between neighboring values), our results show that our techniques answer approximate similarity queries with strong guarantees and an excellent empirical performance, on data series and vectors alike, outperforming the state-of-the-art approximate techniques for vectors when operating on disk, and remaining competitive in memory.

The chapter is organized in seven sections. We summarize our contributions in section 4.1, describe our proposed techniques in 4.3, survey the state-of-the-art approximate techniques in section 4.2, outline our experimental framework and present results and an elaborate discussion in sections 4.4, 4.5 and 4.6, then conclude the chapter in section 4.7.

[1]This chapter is a slightly modified version of [96].

4.1 Main Contributions

Our key contributions are as follows:

1. Following the taxonomy we presented in Chapter 2, we include a brief survey of similarity search approaches supporting approximate search, bringing together works from the data series and high-dimensional data research communities.

2. We propose a new set of approximate approaches with theoretical guarantees on accuracy and excellent empirical performance, based on modifications to the current data series exact methods.

3. We evaluate all methods under a unified framework to prevent implementation bias. We used the most efficient C/C++ implementations available for all approaches, and developed from scratch in C the ones that were only implemented in other programming languages. Our new implementations are considerably faster than the original ones.

4. We conduct the first comprehensive experimental evaluation of the efficiency and accuracy of data series approximate similarity search approaches, using synthetic and real series and vector datasets from different domains, including the two largest vector datasets publicly available.

5. Our results unveil the strengths and weaknesses of each method, and lead to recommendations as to which approach to use.

6. Based on our thorough evaluation, we make an important observation that has never been made in the past: methods derived from the exact data series indexing approaches generally surpass the state-of-the-art techniques for approximate search in vector spaces. This finding paves the way for exciting new developments in the field of approximate similarity search for data series and high-dimensional data at large.

7. We share all source codes, datasets, and queries [94].

4.2 Approaches

Similarity search methods aim at answering a query efficiently by limiting the number of data points accessed, while minimizing the I/O cost of accessing raw data on disk and the CPU cost when comparing raw data to the query (e.g., Euclidean distance calculations). These goals are achieved by exploiting summarization techniques, and using efficient data structures (e.g., an index) and search algorithms. Note that solutions based on sequential scans are geared to exact similarity search [5, 27], and cannot support efficient approximate search, since all candidates are always read.

Answering a similarity query using an index typically involves two steps: a filtering step where the pre-built index is used to prune candidates and a refinement step where the surviving candidates are compared to the query in the original high-dimensional space [40, 41, 20, 28, 30, 42, 25, 43, 44, 31]. Some exact [41, 20, 43, 42] and approximate methods [26, 45] first summarize the original data and then index these summarizations, while others tie together data reduction and indexing [28, 30, 25]. Some approximate methods return the candidates obtained in the filtering step [45]. There also exist exact [46] and approximate [29] methods that index high-dimensional data directly.

A variety of data structures exist for similarity search indexes, including trees [40, 43, 28, 30, 25, 44, 31, 26, 42], inverted indexes [47, 48, 49, 45], filter files [41, 20, 30], hash tables [50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60] and graphs [61, 62, 63, 64, 65, 66, 67, 29]. There also exist multi-step approaches, e.g., Stepwise [24], that transform and organize data according to a hierarchy of resolutions.

Next, we outline the *approximate* similarity search methods (refer also to Table 4.1) and their summarization techniques. (*Exact* methods are detailed in Chapter 3).

Table 4.1: Similarity search methods used in this study ("•" indicates our modifications to original methods). All methods support in-memory data, but only methods ticked in last column support disk-resident data.

		Matching Accuracy				Representation		Implementation		
		exact	ng-appr.	ϵ-appr.	δ-ϵ-appr.	Raw	Reduced	Original	New	Disk-resident Data
Graphs	HNSW		[29]			✓		C++		
	NSG		[168]			✓		C++		
Inv. Indexes	IMI		[45, 159]				OPQ	C++		✓
LSH	QALSH				[60]		Signatures	C++		
	SRS				[26]		Signatures	C++		
Scans	VA+file	[20]	•	•	•		DFT	MATLAB	C	✓
Trees	Flann		[169]			✓		C++		
	DSTree	[25]	[25]	•	•		EAPCA	Java	C	✓
	HD-index		[170]				Hilbert keys	C++		✓
	iSAX2+	[28]	[28]	•	•		iSAX	C#	C	✓

4.2.1 Summarization Techniques

Random projections (used by SRS [26]) reduce the original high-dimensional data into a lower dimensional space by multiplying it with a random matrix. The Johnson-Lindenstrauss (JL) Lemma [171] guarantees that if the projected space has a large enough number of dimensions, there is a high probability that the pairwise distances are preserved, with a distortion not exceeding $(1 + \epsilon)$.

Piecewise Aggregate Approximation (PAA) [154] and *Adaptive Piecewise Constant Approximation* (APCA) [155] are segmentation techniques that approximate a data series S using l segments (of equal/arbitrary length, respectively). The approximation represents each segment with the mean value of its points. The *Extended APCA* (EAPCA) [25] technique extends APCA by representing each segment with both the mean and the standard deviation.

Quantization is a lossy compression process that maps a set of infinite numbers to a finite set of codewords that together constitute the codebook. A *scalar* quantizer operates on the individual dimensions of a vector independently, whereas a *vector* quantizer considers the vector as a whole (leveraging the correlation between dimensions [172]). The size k of a codebook increases exponentially with the number of bits allocated for each code. A *product* quantizer [47] splits the original vector of dimension d into m smaller subvectors, on which a lower-complexity vector quantization is performed. The codebook then consists of the cartesian product of the codebooks of the m subquantizers. Scalar and vector quantization are special cases of product quantization, where m is equal to d and 1, respectively.

(i) *Optimized Product Quantization* (OPQ) (used by IMI [159]) improves the accuracy of the original product quantizer [47] by adding a preprocessing step consisting of a linear transformation of the original vectors, which decorrelates the dimensions and optimizes space decomposition. A similar quantization technique, CK-Means, was proposed in [173] but OPQ is considered the state-of-the-art [174, 175].

(ii) The *Symbolic Aggregate Approximation* (SAX) [156] technique starts by transforming the data series into l real values using PAA, and then applies a *scalar* quantization technique to represent the PAA values using discrete symbols forming an alphabet of size a, called the cardinality of SAX. The l symbols form the SAX representation. The iSAX [157] technique allows comparisons of SAX representations of different cardinalities, which makes SAX indexable.

(iii) *The Karhunen-Loève transform (KLT).* The original VA+file method [20] first converts a data series S of length n using KLT into n real values to de-correlate the data, then applies a *scalar* quantizer to encode the real values as discrete symbols. As we will explain in the next subsection, for efficiency considerations, we altered the VA+file to use the *Discrete Fourier Transform* (DFT) instead of KLT. DFT [120, 124, 151, 152] approximates a data series using l frequency coefficients, and can be efficiently implemented with Fast Fourier Transform (FFT), which is optimal for whole matching (alternatively, the MFT algorithm [153] is adapted to subsequence matching since it uses sliding windows).

4.2.2 Approximate Similarity Search Methods

There exist several techniques for approximate similarity search [50, 158, 176, 177, 21, 178, 179, 180, 26, 159, 29, 90] [181, 182, 183]. We focus on the 7 most prominent techniques designed for high-dimensional data, and we also describe the approximate search algorithms designed specifically for data series. We also propose a new set of techniques that can answer δ-ϵ-approximate queries based on modifications to existing exact similarity methods for data series.

State-of-the-Art for High-dimensional Vectors

Flann [169] is an in-memory ensemble technique for ng-approximate nearest neighbor search in high-dimensional spaces. Given a dataset and a desired search accuracy, Flann selects and auto-tunes the most appropriate algorithm among randomized kd-trees [184]

and a new proposed approach based on hierarchical k-means trees [169].

HD-index [170] is an ng-approximate nearest neighbor technique that partitions the original space into disjoint partitions of lower dimensionality, then represents each partition by an RBD tree (modified B+tree with leaves containing distances of data objects to reference objects) built on the Hilbert keys of data objects. A query Q is partitioned according to the same scheme, searching the hilbert key of Q in the RDB tree of each partition, then refining the candidates first using approximate distances based on triangular and Ptolemaic inequalities then using the real distances.

HNSW. HNSW [29] is an in-memory ng-approximate method that belongs to the class of proximity graphs that exploit two fundamental geometric structures: the Voronoi Diagram (VD) and the Delaunay Triangulation (DT). A VD is obtained when a given space is decomposed using a finite number of points, called *sites*, into regions such that each site is associated with a region consisting of all points that are closer to it than to any other site. The DT is the dual of the VD. It is constructed by connecting sites with an edge if their regions share a side. Since constructing a DT for a generic metric space is not always possible (except if the DT is the complete graph) [185], proximity graphs, which approximate the DT by conserving only certain edges, have been proposed [61, 62, 63, 64, 65, 66, 67, 29]. A k-NN graph is a proximity graph, where only the links to the closest neighbors are preserved. Such graphs suffer from two limitations: (i) the curse of dimensionality; and (ii) the poor performance on clustered data (the graph has a high probability of being disconnected). To address these limitations, the Navigable Small World (NSW) method [65] proposed to heuristically augment the approximation of the DT with long range links to satisfy the small world navigation properties [186]. The HNSW graph [29] improves the search efficiency of NSW by organizing the links in hierarchical layers according to their lengths. Search starts at the top layer, which contains only the longest links, and proceeds down the hierarchy. HNSW is considered the state-of-the-art [187].

NSG [168] is a recent in-memory proximity graph approach that approximates a graph structure called MRNG [168] which belongs to the class of Monotonic Search Networks (MSNET). Building an MRNG graph for large datasets becomes impractical; that is why the state-of-the-art techniques approximate it. NSG approximates the MRNG graph by relaxing the monotonicity requirement and edge selection strategy, and dropping the longest edges in the graph.

IMI. Among the different quantization-based inverted indexes proposed in the literature [47, 48, 49, 45], IMI [159, 45] is considered the state-of-the-art [175]. This class of techniques builds an inverted index storing the list of data points that lie in the proximity of each codeword. The codebook is the set of representative points obtained by performing clustering on the original data. When a query arrives, the ng-approximate search algorithm returns the list of all points corresponding to the closest codeword (or list of codewords).

LSH. The LSH family [188] encompasses a class of randomized algorithms that solve the δ-ϵ-approximate nearest neighbor problem in sub-linear time, for $\delta < 1$. The main intuition is that two points that are nearby in a high-dimensional space, will remain nearby when projected to a lower dimensional space [50]. LSH techniques partition points into buckets using hash functions, which guarantee that only nearby points are likely to be mapped to the same bucket. Given a dataset $\mathbb{S}$ and a query S_Q, L hash functions are applied to all points in $\mathbb{S}$ and to the query S_Q. Only points that fall at least once in the same bucket as S_Q, in each of the L hash tables, are further processed in a linear scan to find the δ-ϵ-approximate nearest-neighbor. There exist many variants of LSH, either proposing different hash functions to support particular similarity measures [51, 52, 53, 58], or improving the theoretical bounds on query accuracy (i.e., δ or ϵ), query efficiency or the index size [54, 55, 56, 57, 58, 59, 26, 60] [189]. In this work, we select SRS [26] and QALSH [60] to represent the class of LSH techniques because they are considered the state-of-the-art in terms of footprint and accuracy, respectively [170]. SRS answers δ-ϵ-

approximate queries using size linear to the dataset size, while empirically outperforming other LSH methods (with size super-linear to the dataset size [51]). QALSH is a query-aware LSH technique that partitions points into buckets using the query as anchor. Other LSH methods typically partition data points before a query arrives, using a random projection followed by a random shift. QALSH, does not perform the second step until a query arrives, thus improving the likelihood that points similar to the query are mapped to the same bucket.

State-of-the-Art for Data Series

While a number of data series methods support approximate similarity search [190, 191, 19, 21, 23, 42, 25, 28, 30], we focus on those that fit the scope of this study, i.e., methods that support out-of-core k-NN queries with Euclidean distance. In particular, we examine DSTree [25], iSAX2+ [28], and VA+file [20], the three data series methods that perform the best in terms of exact search [95], and also inherently support ng-approximate search.

DSTree [25] is a tree index based on the EAPCA summarization technique and supports ng-approximate and exact query answering. Its dynamic segmentation algorithm allows tree nodes to split vertically and horizontally, unlike the other data series indexes which allow either one or the other. The DSTree supports a lower and upper bounding distance and uses them to calculate a QoS measure that determines the optimal way to split any given node. We significantly improved the efficiency of the original DSTree Java implementation by developing it from scratch in C and optimizing its buffering and memory management, making it 4 times faster across datasets ranging between 25-250GB.

SAX-based indexes include different flavors of tree indexes based on SAX summarization [164]. The original iSAX index [146] was enhanced with a better spliting policy and bulk-loading support in iSAX 2.0 [23], while iSAX2+ [28] further optimized bulk-loading. ADS+ [30] then improved upon iSAX2+ by making it adaptive, Coconut [192, 193, 34]

by constructing a compact and contiguous data layout, and DPiSAX [44, 33], ParIS [32] and MESSI [38] by exploiting parallelization. Here, we use iSAX2+, because of its excellent performance [95] and the fact that the SIMS query answering strategy [30] of ADS+, Coconut, and ParIS is not immediately amenable to approximate search with guarantees (we plan to extend these methods in our future work). We do not include DPiSAX and MESSI, because they are distributed, and in-memory only, algorithms, respectively.

TARDIS [194] is a distributed indexing method that supports exact and ng-approximate kNN queries. It improves the efficiency and accuracy of iSAX by building a more compact, k-ary tree index, exploiting word-level (instead of character-level) cardinality, and using a novel conversion scheme between SAX representations. We do not include TARDIS in the experimental evaluation since it is a distributed algorithm (built in Scala for Spark).

VA+file [20] is a skip-sequential method that improves the accuracy and efficiency of the VA-file [41]. Both techniques create a file that contains quantization-based summarizations of the original high-dimensional data. Search proceeds by sequentially reading each summarization, calculating its lower bounding distance to the query, and accessing the original high-dimensional vector only if the lower bounding distance is less than the current *best-so-far (bsf)* answer. We greatly improved the performance of the original VA+file by approximating KLT with DFT [20, 163] and implementing it in C instead of Matlab. In the rest of the text, whenever we mention the VA+file, we refer to the modified version.

4.3 A New Class of Approximate Search Techniques

We now propose extensions to the data series methods described above, that will allow them to support ϵ-approximate and δ-ϵ-approximate search (in addition to ng-approximate search that they already support). For brevity, we only discuss the tree-

Algorithm 1 exactNN($\boldsymbol{S_Q}$,$\boldsymbol{idx}$)

```
1: bsf.dist ← ∞ ; bsf.node ← NULL;
2: for each rootNode in idx do
3:     result.node ← rootNode;
4:     result.dist ← calcMinDist(S_Q,rootNode);
5:     push result to pqueue
6: bsf ← ng-approxNN(S_Q,idx);
7: add bsf to pqueue;
8: while result ← pop next node from pqueue do
9:     n ← result.node;
10:    if n.dist > bsf.dist then break;
11:    if n is a leaf then                                  ▷ a leaf node
12:        for each S_C in n do
13:            realDist ← calcRealDist(S_Q,S_C);
14:            if realDist < bsf.dist then
15:                bsf.dist ← realDist ;
16:                bsf.node ← n;
17:    else                                                 ▷ an internal node
18:        for each childNode in n do
19:            minDist ← calcMinDist(S_Q,childNode);
20:            if minDist < bsf.dist then add childNode to
21:                pqueue with priority minDist;
22: return bsf
```

based methods (such as iSAX2+ and DSTree); skip-sequential techniques (such as VA+file) can be modified following the same ideas.

The exact 1-NN search algorithms of DSTree and iSAX2+ are based on an optimal exact NN algorithm first proposed for PMR-Quadtree [195], which was then generalized for any hierarchical index structure that is constructed using a conservative and recursive partitioning of the data [196].

Algorithm 1 describes an index-invariant algorithm for exact 1-NN search. It takes as arguments a query S_Q and an index idx. Lines 1-5 initialize the *best-so-far (bsf)* answer and a priority queue with the root node(s) of the index in increasing order of lower bounding (*lb*) distances (the *lb* distance is calculated by the function $calcMinDist$). In line 6, the ng-approxNN function traverses one path of the index tree visiting one leaf to return an ng-approximate bsf answer, which is added to the queue (line 7). In line 8, the algorithm pops nodes from the queue, terminating in line 10 if the *lb* distance of the current node is greater than the current *bsf* distance (the *lb* distances of all remaining nodes in the queue are also greater than the *bsf*). Otherwise, if the node is a leaf, the *bsf* is updated if a better answer is found (lines 11-16); if the node is an internal node,

Algorithm 2 deltaEpsilonNN(S_Q,idx,δ,ϵ, $F_Q(.)$)

```
1: bsf.dist ← ∞ ; bsf.node ← NULL;
   r_δ(Q) ← calcDeltaRadius(S_Q,δ, F_Q(.));
2: bsf ← ng-approxNN(S_Q,idx);
3: add bsf to pqueue;
4: for each rootNode in idx do
5:     result.node ← rootNode;
6:     result.dist ← calcMinDist(S_Q,rootNode);
7:     push result to pqueue
8: while result ← pop next node from pqueue do
9:     n ← result.node;
10:    if n.dist > bsf.dist/(1 + ε) then break;
11:    if n is a leaf then                                    ▷ a leaf node
12:        for each S_C in n do
13:            realDist ← calcRealDist(S_Q,S_C);
14:            if realDist < bsf.dist then
15:                bsf.dist ← realDist ;
16:                bsf.node ← n;
                   if bsf.dist ≤ (1 + ε) r_δ(Q) then exit;
17:    else                                                   ▷ an internal node
18:        for each childNode in n do
19:            minDist ← calcMinDist(S_Q,childNode);
20:            if minDist < bsf.dist/(1 + ε) then add
21:                childNode to pqueue with priority minDist;
22: return bsf
```

its children are added to the queue provided their lb distances are greater than the bsf distance (lines 18-21).

We can use Algorithm 1 for ng-approximate search, by visiting one leaf and returning the first bsf. This ng-approximate answer can be anywhere in the data space

We extend approximate search in Algorithm 1 by introducing two changes: (i) allow the index to visit up to *nprobe* leaves (user parameter); and (ii) apply the modifications suggested in [18] to support δ-ϵ-approximate NN search. The first change is straightforward, so we only describe the second change in Algorithm 2. To return the ϵ-approximate NN of S_Q, S_ϵ, $\boldsymbol{bsf.dist}$ is replaced with $\boldsymbol{bsf.dist}/(1+\epsilon)$ in lines 10 and 20. To return the δ-ϵ-approximate NN of S_Q, $S_{\delta\epsilon}$, we also modify lines 1 and 16.

The distance $r_\delta(Q)$ is initialized in line 1 using $F_Q(\cdot)$, S_Q and δ. $F_Q(\cdot)$ represents the relative distance distribution of S_Q. Intuitively, $r_\delta(Q)$ is the maximum distance from S_Q, such that the sphere with center S_Q and radius $r_\delta(Q)$ is empty with probability δ. As proposed in [197], we use $F(\cdot)$, the overall distance distribution, instead of $F_Q(\cdot)$ to estimate $r_\delta(Q)$. The delta radius $r_\delta(Q)$ is then used in line 16 as a stopping condition.

When $\delta = 1$, Algorithm 2 returns $S_{\delta\epsilon}$, the ϵ-approximate NN of S_Q, and when $\delta = 1$ and $\epsilon = 0$, Algorithm 2 becomes equivalent to Algorithm 1, i.e., it returns S_x, the exact NN of S_Q. Our implementations generalize Algorithm 2 to the case of $k \geq 1$. These modifications are straightforward and omitted for the sake of brevity. A proof of correctness for Algorithm 2 can be found in [18, 198] for $k = 1$ and $k \geq 1$, respectively.

4.4 Experimental Evaluation

We assessed all methods on the same framework. Source code, datasets, queries, and all results are available in [94].

4.4.1 Environment.

All methods were compiled with GCC 6.2.0 under Ubuntu Linux 16.04.2 with their default compilation flags; optimization level was set to 2. Experiments were run on a server with two Intel Xeon E5-2650 v4 2.2GHz CPUs, 75GB[2] of RAM, and 10.8TB (6 x 1.8TB) 10K RPM SAS hard drives in RAID0 with a throughput of 1290 MB/sec.

4.4.2 Experimental Setup

Algorithms. We use the most efficient C/C++ implementation available for each method: iSAX2+ [93], DSTree [93] and VA+file [93] representing exact data series methods with support for approximate queries; and HNSW [199], Faiss IMI [35], SRS [200], FLANN [169], and QALSH [60] representing strictly approximate methods for vectors. We ran experiments with the HD-index [170] and NSG [168], but since they could not scale for our smallest 25GB dataset, we do not report results for them. We extended DSTree, iSAX2+ and VA+file with Algorithm 2, approximating r_δ with density his-

[2]We used GRUB to limit the amount of RAM, so that all methods are forced to use the disk. Note that GRUB prevents the operating system from using the rest of the RAM as a file cache, which is what we wanted for our experiments.

tograms on a 100K data series sample, following the C++ implementation of [18]. All methods are single core implementations, except for HNSW and IMI that make use of multi-threading and SIMD vectorization. Data series points are represented using single precision values and methods based on fixed summarizations use 16 dimensions.

Datasets. We use synthetic and real datasets. Synthetic datasets, called *Rand*, were generated as random-walks using a summing process with steps following a Gaussian distribution (0,1). Such data model financial time series [124] and have been widely used in the literature [124, 28, 135]. Our four real datasets cover domains as varied as deep learning, computer vision, seismology, and neuroscience. *Deep1B* [167] comprises 1 billion vectors of size 96 extracted from the last layers of a convolutional neural network. *Sift1B* [48, 201] consists of 1 billion SIFT vectors of size 128 representing image feature descriptions. To the best of our knowledge, these two vector datasets are the largest publicly available real datasets. *Seismic100GB* [140], contains 100 million data series of size 256 representing earthquake recordings at seismic stations worldwide. *Sald100GB* [166] contains neuroscience MRI data and includes 200 million data series of size 128. In our experiments, we vary the size of the datasets from 25GB to 250GB. The name of each dataset is suffixed with its size. We do not use other real datasets that have appeared in the literature [202, 187], because they are very small, not exceeding 1GB in size.

Queries. All our query workloads consist of 100 query series run asynchronously, i.e., not in batch mode. Synthetic queries were generated using the same random-walk generator as the *Rand* dataset (with a different seed, reported in [94]). For the Deep1B and Sift1B datasets, we randomly select 100 queries from the real workloads that come with the datasets archives. For the other real datasets, query workloads were generated by adding progressively larger amounts of noise to data series extracted from the raw data, so as to produce queries having different levels of difficulty, following the ideas in [149]. Our experiments cover ng-approximate and δ-ϵ-approximate k-NN queries, where k $\in [1, 100]$. We also include results for exact queries to serve as a yardstick.

Scenarios. Our experimental evaluation proceeds in four main steps: (i) we tune methods to their optimal parameters (§5.5.1); (ii) we evaluate the indexing scalability of the methods (§4.5.2); (iii) we compare in-memory and out-of-core scalability and accuracy of all methods (§4.5.3-§4.5.4); and (iv) we perform additional experiments on the best performing methods for disk-resident data (§4.5.4).

Measures. We assess methods using the following criteria:

(1) Scalability and search efficiency using: *wall clock time* (input, output, CPU and total time), *throughput* (# of queries answered per minute), and two implementation-independent measures: the *number of random disk accesses* (# of disk seeks) and the *percentage of data accessed*.

(2) Search accuracy is assessed using: *Avg_Recall*, *Mean Average Precision (MAP)*, and *Mean Relative Error (MRE)*. Recall is the most commonly used accuracy metric in the approximate similarity search literature. However, since it does not consider rank accuracy, we also use MAP [203] that is popular in information retrieval [204, 205] and has been proposed recently in the high-dimensional community [170] as an alternative accuracy measure to recall. For a workload of queries $S_{Q_i} : i \in [1, N_Q]$, these are defined as follows.

1. $Avg_Recall(workload) = \sum_{i=1}^{N_Q} Recall(S_{Q_i})/N_Q$

2. $MAP(workload) = \sum_{i=1}^{N_Q} AP(S_{Q_i})/N_Q$

3. $MRE(workload) = \sum_{i=1}^{N_Q} RE(S_{Q_i})/N_Q$

where:

- $Recall(S_{Q_i}) = \frac{\#\ true\ neighbors\ returned\ by\ Q_i}{k}$
- $AP(S_{Q_i}) = \frac{\sum_{r=1}^{k}(P(S_{Q_i},r) \times rel(r))}{k}, \forall i \in [1, N_Q]$
  - $P(S_{Q_i}, r) = \frac{\#\text{ true neighbors among the first } r \text{ elements}}{r}$.
  - $rel(r)$ is equal 1 if the neighbor returned at position r is one of the k exact neighbors of S_{Q_i} and 0 otherwise.

- $RE(S_{Q_i}) = \frac{1}{k} \times \sum_{r=1}^{k} \frac{d(S_{Q_i}, S_{C_r}) - d(S_{Q_i}, S_{C_i})}{d(S_{Q_i}, S_{C_i})}$. S_{C_i} is the exact nearest neighbor of S_{Q_i} and S_{C_r} is the r-th NN retrieved[3]. Without loss of generality, we do not consider the case where $d(S_{Q_i}, S_{C_i}) = 0$. (i.e., range queries with radius zero, or kNN queries where the 1-NN is the query itself[4].)

(3) Size, using the *main memory* footprint of the algorithm.

Procedure. Experiments involve two steps: index building and query answering. Caches are fully cleared before each step, and stay warm between consecutive queries. For large datasets that do not fit in memory, the effect of caching is minimized for all methods. All experiments use workloads of 100 queries. Results reported for workloads of 10K queries are extrapolated: we discard the 5 best and 5 worst queries of the original 100 (in terms of total execution time), and multiply the average of the 90 remaining queries by 10K.

4.5 Results

4.5.1 Parametrization

We start by fine tuning each method (graphs omitted for brevity). In order to understand the speed/accuracy tradeoffs, we fix the total memory size available to 75GB. The optimal parameters for DSTree, iSAX2+ and VA+file are set according to [95]. For indexing, the buffer and leaf sizes are set to 60GB and 100K, respectively, for both DSTree and iSAX2+. iSAX2+ is set to use 16 segments. VA+file uses a 20GB buffer and 16 DFT symbols. For SRS, we set M (the projected space dimensionality) to 16 so that the representations of all datasets fit in memory. The settings were the same for all datasets. The fine tuning for HNSW and IMI proved more tricky and involved many testing iterations since the index building parameters strongly affect the speed/accuracy of query answering and differ greatly across datasets. For this reason, different parameters were chosen for

[3]Note that in Definition 9, ϵ is an upper bound on $RE(S_{Q_i})$.

[4]In these cases, the MRE definition can be extended to use the symmetric mean absolute percentage error [206].

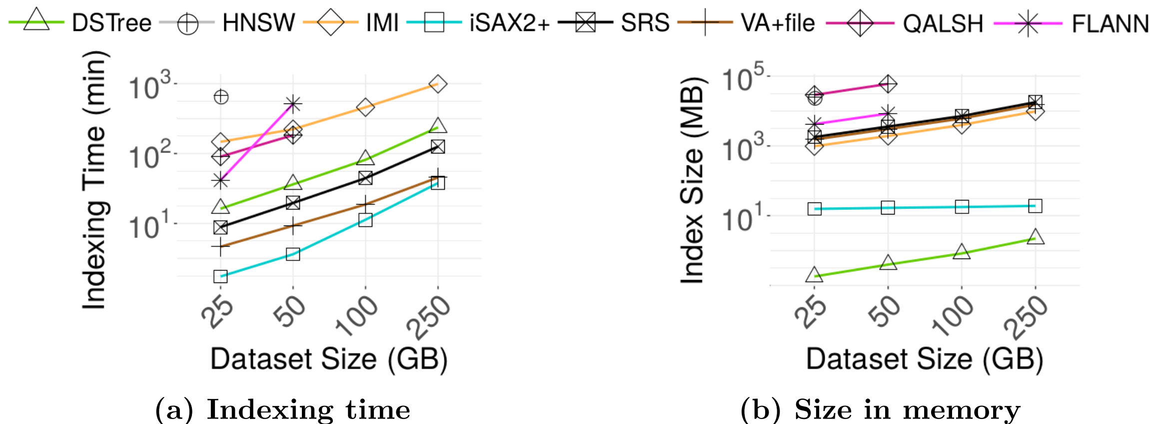
different datasets. For the in-memory method HNSW, we set efConstruction (the number of neighbors considered during index construction) to 500, and M (the number of bi-directional edges created for every new node during indexing) to 4 for the Rand25GB dataset. For Deep25GB and Sift25GB, we set efConstruction to 500 and M to 16. To tune the Faiss implementation of IMI, we followed the guidelines in [35]. For the in-memory datasets, we set the index factory key to PQ32_128,IMI2x12,PQ32 and the training size to 1,048,576 vectors, while for disk based datasets, the index key is PQ32_128,IMI2x14,PQ32 and the training size is 4,194,304 vectors. To tune δ-ϵ-approximate search performance and accuracy, we vary δ and ϵ for SRS and ϵ for DSTree, iSAX2+ and VA+file (except in one experiment where we also vary δ). For ng-approximate search, we vary the *nprobe* parameter for DSTree/iSAX2+/IMI/VA+file (*nprobe* represents the number of visited leaves for DSTree/iSAX2+, the number of visited raw series for VA+file, and the number of inverted lists for IMI), and the *efs* parameter for HNSW (which represents the number of non-pruned candidates).

Since none of the methods supports auto-tuning, we perform this step manually. Ideally, parametrization would be performed exhaustively for each scenario used in the experimental study. However, this will require an excessive amount of resources and time as a scenario is defined by 4 inputs: an algorithm, a dataset, a dataset size and a data series length. For instance, parameter tuning for the HNSW method on the small Deep25GB alone took 120 hours. Developing auto-tuning mechanisms for these methods is an open problem which we would like to explore in future work.

4.5.2 Indexing Efficiency

In this section, we evaluate the indexing scalability of each method by varying the dataset size. We used four synthetic datasets of sizes 25GB, 50GB, 100GB and 250GB, two of which fit in memory (total RAM was 75GB).

Figure 4.1a shows that iSAX2+ is the fastest method at index building in and out of memory, followed by VA+file, SRS, DSTree, FLANN, QALSH, IMI and HNSW. Even though IMI and HNSW are the only parallel methods, they are the slowest at index building. Although FLANN is slow at indexing the 50GB dataset, we think this is more due to memory management issues in the code, which cause swapping. For HNSW, the major cost is building the graph structure, whereas IMI spends most of the time on determining the clusters and computing the product quantizers. We also measured the breakdown of the indexing time and found out that all methods can be significantly improved by parallelism except iSAX2+ and QALSH that are I/O bound. In terms of footprint, the DSTree is the most memory-efficient, followed by iSAX2+. IMI, SRS, VA+file and FLANN are two orders of magnitude larger, while QALSH and HNSW are a further order of magnitude bigger (Figure 4.1b).



(a) Indexing time **(b) Size in memory**

Figure 4.1: Comparison of indexing scalability

4.5.3 Query Answering Efficiency and Accuracy: in-Memory Datasets

We now compare query answering efficiency and accuracy, in addition to the indexing time, thus, measuring how well each method amortizes index construction time over a large number of queries, and the level of accuracy achieved.

Summary. For our in-memory experiments, we used four datasets of 25GB each: two synthetic (with series of length 256 and 16384, respectively), and two real: Deep25GB and Sift25GB. We ran 1NN, 10NN and 100NN queries on the four datasets and we observed that, while the running times increase with k, the relative performance of the methods stays the same. Due to lack of space, Figures 4.2 and 4.3 show the 100NN query results only (full results are in [94]), which we discuss below. Note that HNSW, QALSH and FLANN store all raw data in-memory, while all other approaches use the memory to store their data structures, but read the raw data from disk; IMI does not access the raw data at all (it only uses the in-memory summaries).

Short Series. For ng-approximate queries of length 256 on the Rand25GB dataset, HNSW has the largest throughput for any given accuracy, followed by FLANN, IMI, DSTree and iSAX2+ (Figure 4.2a). However, HNSW does not reach a MAP of 1, which is only obtained by the data series indexes (DSTree, iSAX2+, VA+file). The skip-sequential method VA+file performs poorly on approximate search since it prunes per series and not per cluster like the tree-based methods do. When indexing time is also considered, iSAX2+ wins for the workload consisting of 100 queries (Figure 4.2b), and DSTree for the 10K queries (Figure 4.2c).

Regarding δ-ϵ-approximate search, DSTree offers the best throughput/accuracy trade-off, followed by iSAX2+, SRS, VA+file and finally QALSH. SRS does not achieve a MAP higher than 0.5, while DSTree and iSAX2+ are at least 3 times faster than SRS for a similar accuracy (Figure 4.3a). When we consider the combined indexing and querying times, iSAX2+ wins over all methods for 100 queries (Figure 4.3b), and DSTree wins for 10K queries (Figure 4.3c).

Long Series. In this experiment, we use dataset sizes of 25GB, and query length of 16384. For ng-approximate search, we report the results only for iSAX2+, DSTree and VA+file. We ran several experiments with IMI and HNSW building the indexes using different parameters, but obtained a MAP of 0 for IMI for all index configurations we

tried, and ran into a segmentation fault during query answering with HNSW. DSTree outperforms both iSAX2+ and VA+file in terms of throughput and combined total cost for the larger workload (Figures 4.2d and 4.2f), whereas iSAX2+ wins for the smaller workload when the combined total cost is considered (Figure 4.2e). We note also that the performance of FLANN deteriorates with the increased dimensionality.

For δ-ϵ-approximate queries, Figure 4.3d shows that DSTree and VA+file outperform all other methods for large MAP values, while DSTree and iSAX2+ have higher throughput for small MAP values. Note that the SRS accuracy decreases when compared to series of length 256, with the best MAP value now being 0.25. This is due to the increased information loss, as for both series lengths the number of dimensions in the projected space is 16. When index building time is considered, VA+file wins for the small workload (Figure 4.3e), and iSAX2+ and DSTree win for the large one (Figure 4.3f). We do not report numbers for QALSH because the algorithm ran into a segmentation fault for series of length 16384.

Real Data. We ran the same set of experiments with real datasets. For *ng*-approximate queries, HNSW outperforms the query performance of other methods by a large margin (Figures 4.2g and 4.2j). When indexing time is considered, HNSW loses its edge due to its high indexing cost to iSAX2+ when the query workload consists of 100 queries (Figures 4.2h and 4.2k) and to DSTree for the 10K workload (Figures 4.2i and 4.2l). HNSW does not achieve a MAP of 1, while DSTree and ISAX2+ both do, yet at a high cost.

DSTree clearly wins on Sift25GB and Deep25GB among δ-ϵ-approximate methods (Figures 4.3g, 4.3j, 4.3i, and 4.3l), except for the scenario of indexing plus answering 100 queries, where iSAX2+ has the least combined cost (Figures 4.3h and 4.3k). This is because DSTree's query answering is very fast, but its indexing cost is high, so it is amortized only with a large query workload (Figures 4.3i and 4.3l). We observe a similar trend for both Sift25GB and Deep25GB, except the degradation of the performance of

SRS, which achieves a very low accuracy of 0.01 on Deep25GB, despite using the most restrictive parameters ($\delta = 0.99$ and $\epsilon = 0$).

Comparison of Accuracy Measures. In the approximate similarity search literature, the most commonly used accuracy measures are approximation error and recall. The approximation error evaluates how far the approximate neighbors are from the true neighbors, whereas recall assesses how many true neighbors are returned. In our study, we refer to the recall and approximation error of a workload as Avg_Recall and MRE respectively. In addition, we use a third measure called MAP because it takes into account the order of the returned candidates and thus is more sensitive than recall. Figures 4.4a and 4.4b compare all three measures for the popular real dataset Sift25GB (we use the 25GB subset to include in-memory methods as well). We observe that for any given workload, the Avg_Recall is equal to MAP for all methods, except for IMI. This is because IMI returns the short-listed candidates based on distance calculations on the compressed vectors, while the other methods further refine the candidates by sorting them based on the Euclidean distance of the query to the raw data. Figure 4.4b illustrates the relationship between MAP and MRE. Note that the value of the approximation error is not always indicative of the actual accuracy. For instance, an MRE of about 0.5 for iSAX2 sounds acceptable (some popular LSH methods only work with $\epsilon >= 3$ [22, 58]), yet it corresponds to a very low accuracy of 0.03 as measured by MAP (Figures 4.4b). Note that MAP can be more useful in practice, since it takes into account the actual ranks of the true neighbors returned, whereas MRE is evaluated only on the distances between the query and its neighbors.

4.5.4 Query Answering Efficiency and Accuracy: on-Disk Datasets

We now report results (Figures 4.5 and 4.6) for on-disk experiments, excluding the in-memory only HNSW, QALSH and FLANN.

Synthetic Data. DSTree and iSAX2+ outperform by far the rest of the techniques on both ng-approximate and δ-ϵ-approximate queries. iSAX2+ is particularly competitive when the total cost is considered with the smaller workload (Figures 4.5b and 4.6b). The querying performance of SRS degraded on-disk due to severe swapping issues (Figure 4.6a), therefore we do not include this method in further disk-based experiments. Although IMI is much faster than both iSAX2+ and DSTree on ng-approximate search, its accuracy is extremely low. In fact, the best MAP accuracy achieved by IMI plummets to 0.05, whereas DSTree and iSAX2+ have much higher MAP values (Figure 4.5a).

Real Data. DSTree outperforms all methods on both Sift250GB and Deep250GB. The only exception is iSAX2+ having an edge when the combined indexing and search costs are considered for the smaller workload (Figures 4.5e, 4.6e, 4.5h and 4.6h) and being equally competitive on ng-approximate query answering (Figures 4.5d, 4.6d).

Best Performing Methods. The earlier results show that VA+file is outperformed by DSTree and iSAX2+, and that SRS and IMI have very low accuracy on the large datasets. We thus conduct further experiments considering only iSAX2+ and DSTree (recall that HNSW is an in-memory approach only): see Figures 4.7, 4.8 and 4.9.In terms of query efficiency/accuracy tradeoff, DSTree outperforms iSAX2+ on all datasets, except for Sald100GB (Figure 4.7j), and for low MAP values on Seismic100GB (Figure 4.7m).

Amount of data accessed. As expected, both DSTree and iSAX2+ need to access more data as the accuracy increases. Nevertheless, we observe that to achieve accuracies of almost 1, both methods access close to 100% of the data for Sift250GB (Figure 4.7e), Deep250GB (Figure 4.7h) and Seismic100GB (Figure 4.7n), compared to 10% of data accessed on Sald100GB (Figure 4.7k) and Rand250GB. (Figure 4.7b). The percentage of accessed data also varies among real datasets, Deep250GB and Sift250GB requiring the most. Note that for some datasets, a MAP of 1 is achievable with minimal data access. For instance DSTree needs to access about 1% of the data to get a MAP of 1 on Sald100GB (Figure 4.7k).

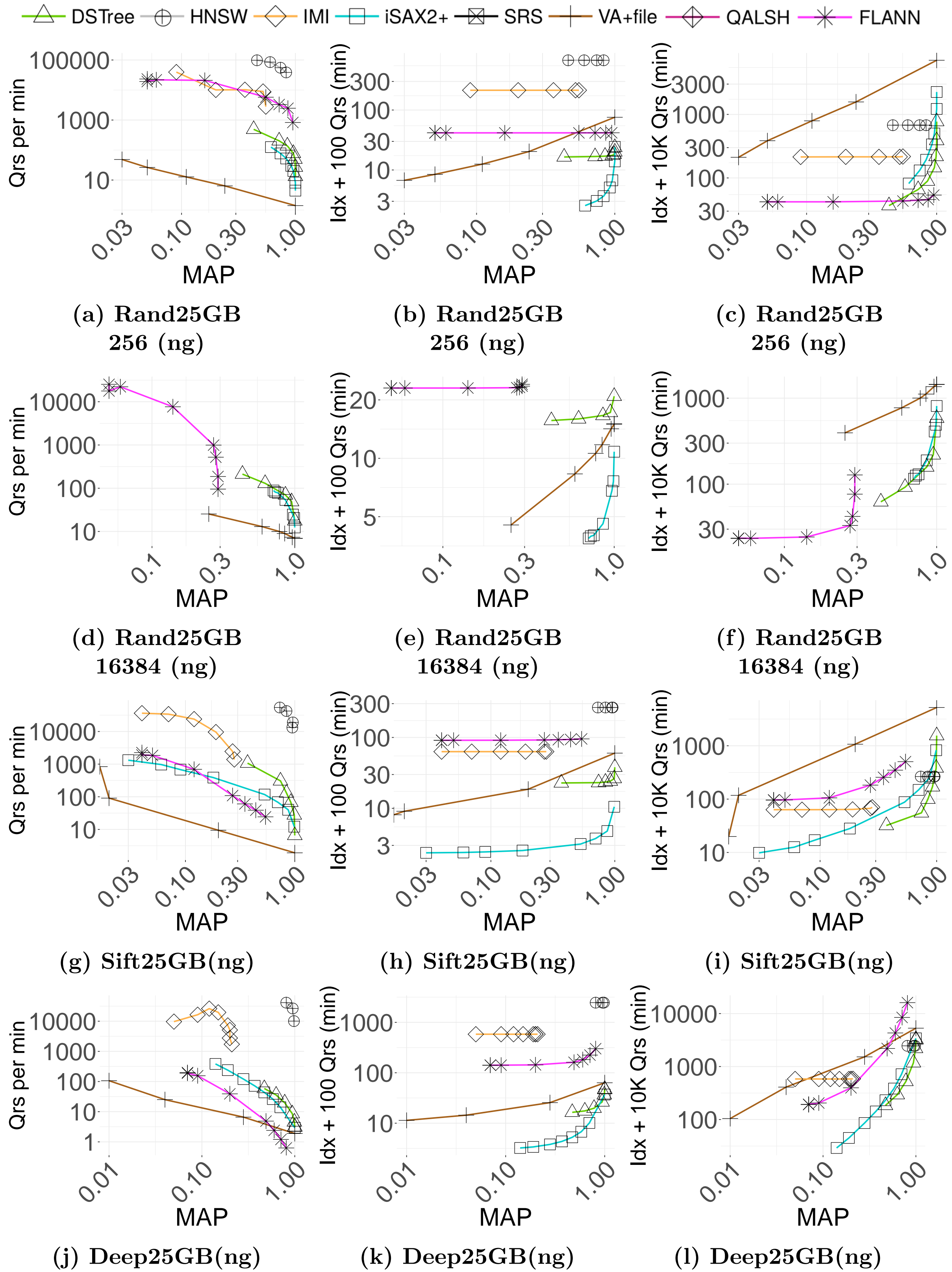
Number of Random I/Os. To understand the nature of the data accesses discussed above, we report the number of random I/Os in Figure 4.7 (bottom row). Overall, iSAX2+ incurs a higher number of random I/Os for all datasets. This is because iSAX2+ has a larger number of leaves, with a smaller fill factor than DSTree [95]. For instance, the large number of random I/Os incurred by iSAX2+ (Figure 4.7o) is what explains the faster runtime of DSTree on the Seismic100GB dataset (Figure 4.7m), even if DSTree accesses more data than iSAX2+ for higher MAP values (Figure 4.7n). The Sald100GB dataset is an exception to this trend as iSAX2+ outperforms DSTree on all accuracies except for MAP is 1 (Figure 4.7j), because it accesses less data incuring almost the same random I/O (Figures 4.7k and 4.7l).

Effect of k. Figure 4.8 summarizes experiments varying k on different datasets in-memory and on-disk. We measure the total time required to complete a workload of 100 queries for each value of k. We observe that finding the first neighbor is the most costly operation, while finding the additional neighbors is much cheaper.

Effect of δ and ϵ. In Figure 4.9, we describe in more detail how varying δ and ϵ affects the performance of DSTree and iSAX2+. Figure 4.9a shows that the throughput of both methods increases dramatically with increasing ϵ. For example, a small value of $\epsilon = 5$ increases the throughput of iSAX2+ by two orders of magnitude, when compared to exact search ($\epsilon = 0$). Moreover, note that both methods return the actual exact answers for small ϵ values, and accuracy drops only as ϵ goes beyond 2 (Figure 4.9b). In addition, Figure 4.9c shows that the actual approximation error MRE is well below the user-tolerated threshold (represented by ϵ), even for ϵ values well above 2. The above observations mean that these methods can be used in approximate mode, achieving very high throughput, while still returning answers that are exact (or very close to the exact).

As the probability δ increases, throughput stays constant and only plummets when search becomes exact ($\delta = 1$ in Figure 4.9d). Similarly, accuracy also stays constant, then slightly increases (for a very high δ of 0.99), reaching 1 for exact search (Figure 4.9e).

Accuracy plateaus as δ increases, because the first ng-approximate answer found by both algorithms is very close to the exact answer (Figures 4.9b and 4.9c) and better than the approximation of r_δ, thus the stopping condition is never triggered. When a high value of δ is used, the stopping condition takes effect for some queries, but the runtime is very close to that of the exact algorithm.



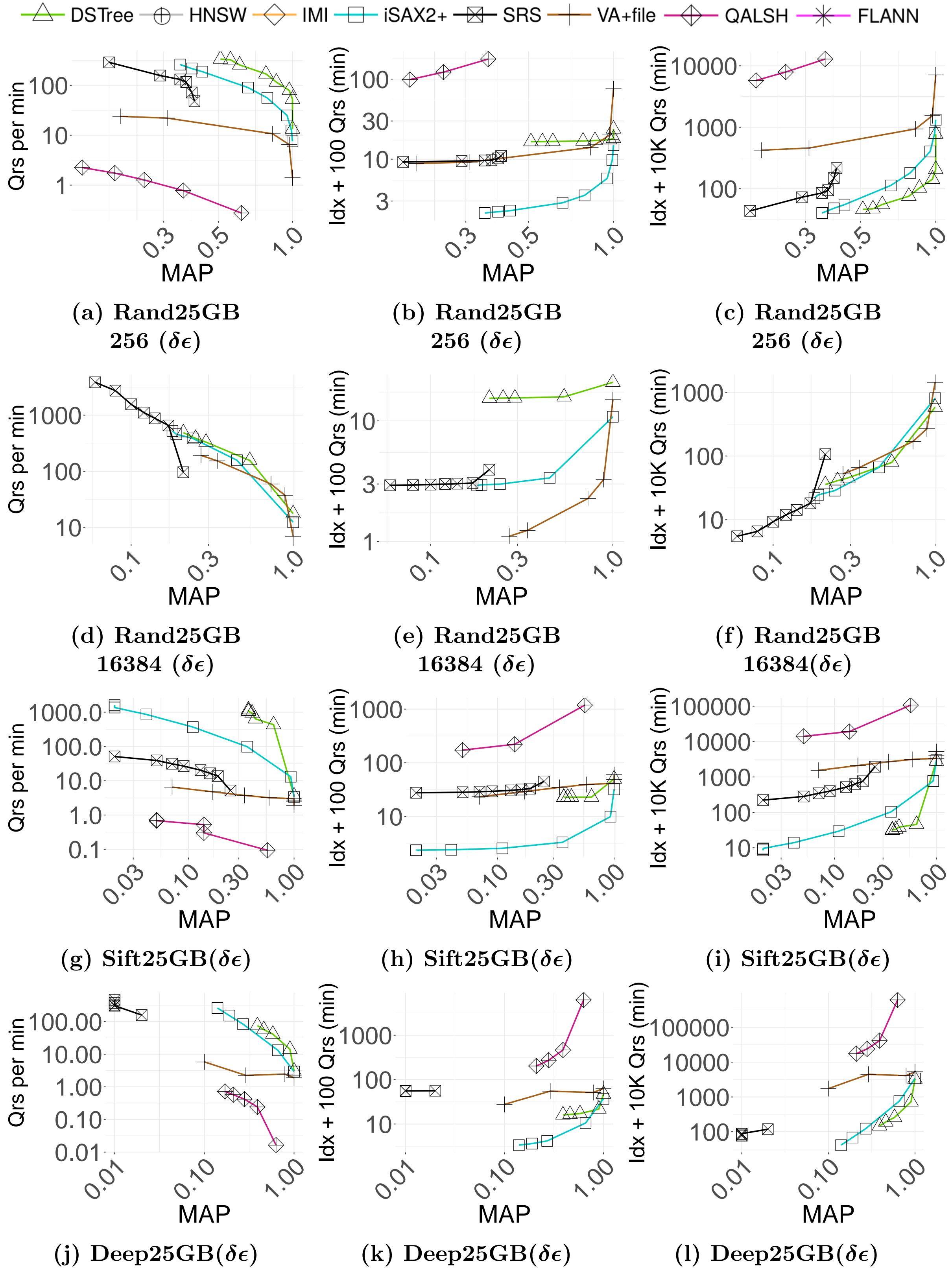
(a) **Rand25GB 256 (ng)** (b) **Rand25GB 256 (ng)** (c) **Rand25GB 256 (ng)**

(d) **Rand25GB 16384 (ng)** (e) **Rand25GB 16384 (ng)** (f) **Rand25GB 16384 (ng)**

(g) **Sift25GB(ng)** (h) **Sift25GB(ng)** (i) **Sift25GB(ng)**

(j) **Deep25GB(ng)** (k) **Deep25GB(ng)** (l) **Deep25GB(ng)**

Figure 4.2: Efficiency vs. accuracy for in memory *ng*-approximate search (100NN queries)



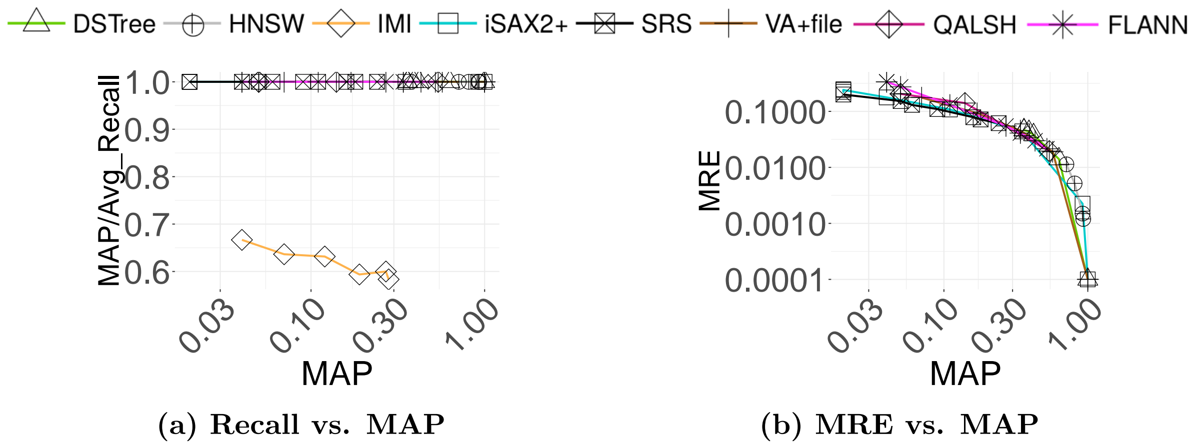
(a) **Rand25GB 256 ($\delta\epsilon$)**
(b) **Rand25GB 256 ($\delta\epsilon$)**
(c) **Rand25GB 256 ($\delta\epsilon$)**

(d) **Rand25GB 16384 ($\delta\epsilon$)**
(e) **Rand25GB 16384 ($\delta\epsilon$)**
(f) **Rand25GB 16384($\delta\epsilon$)**

(g) **Sift25GB($\delta\epsilon$)**
(h) **Sift25GB($\delta\epsilon$)**
(i) **Sift25GB($\delta\epsilon$)**

(j) **Deep25GB($\delta\epsilon$)**
(k) **Deep25GB($\delta\epsilon$)**
(l) **Deep25GB($\delta\epsilon$)**

Figure 4.3: Efficiency vs. accuracy for in-memory δ-ϵ-approximate search(100NN queries)



(a) Recall vs. MAP **(b) MRE vs. MAP**

Figure 4.4: Comparison of measures (Sift25GB)

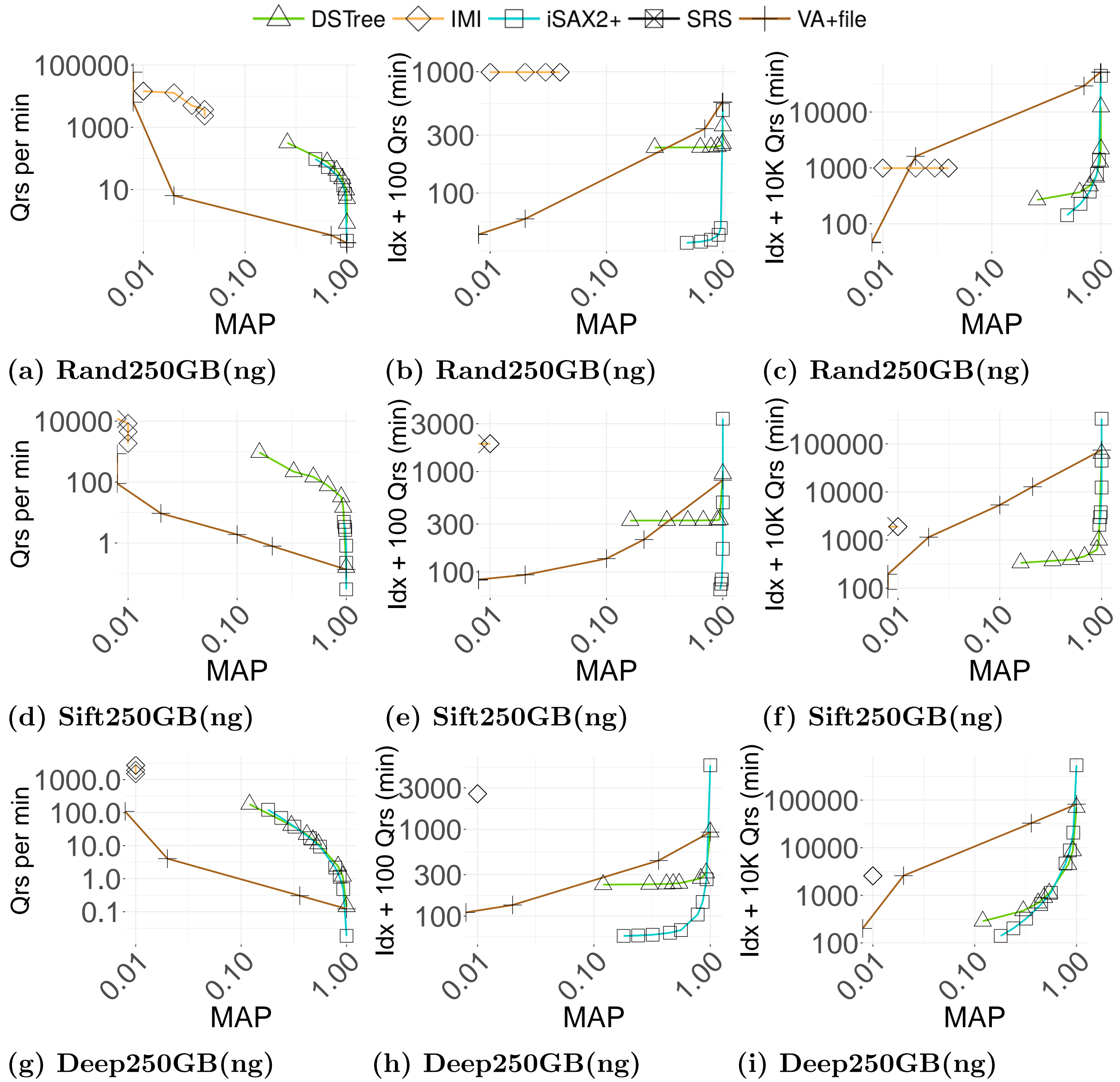
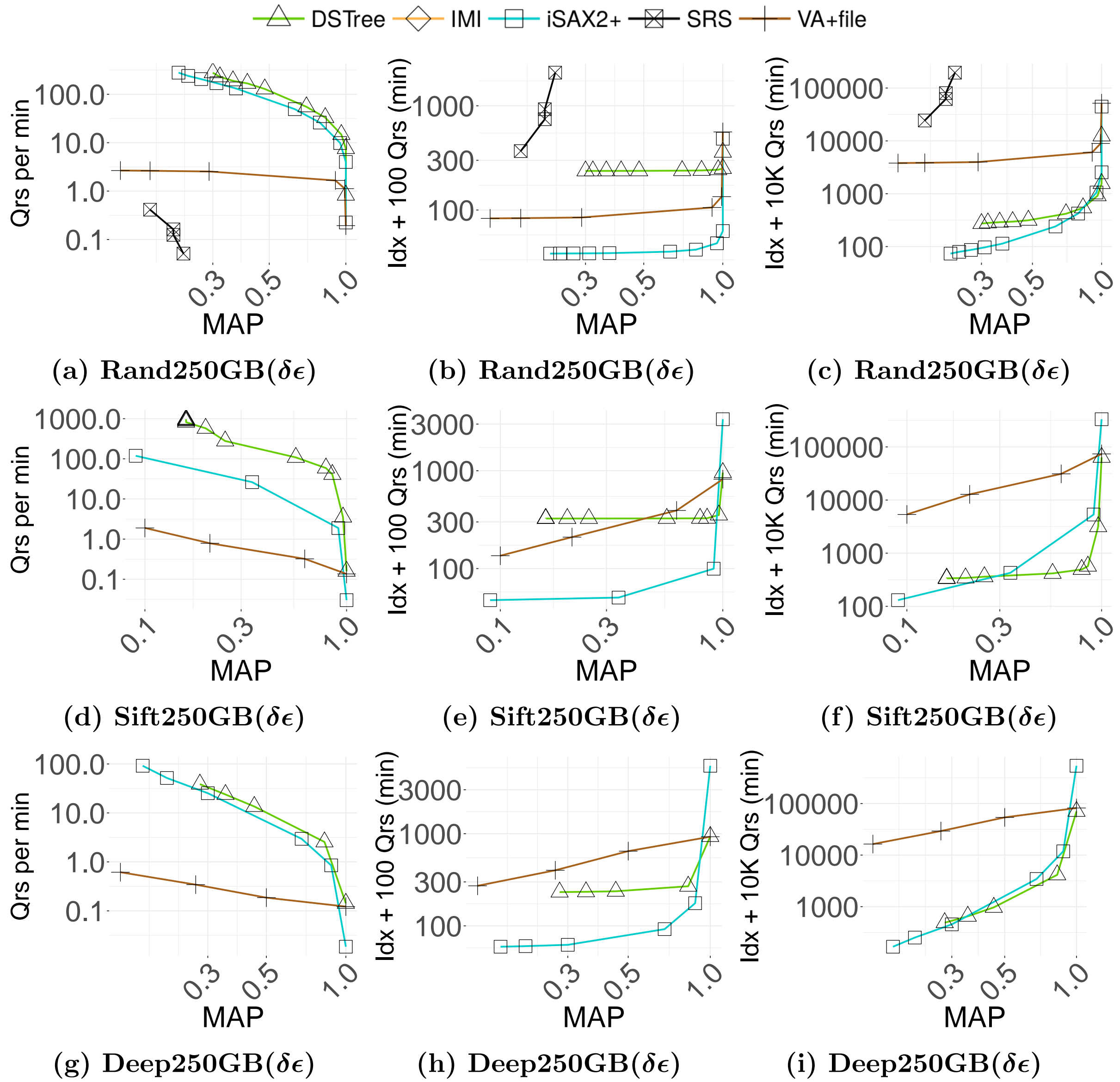


Figure 4.5: Efficiency vs. accuracy for on disk ng-approximate search (100NN queries)

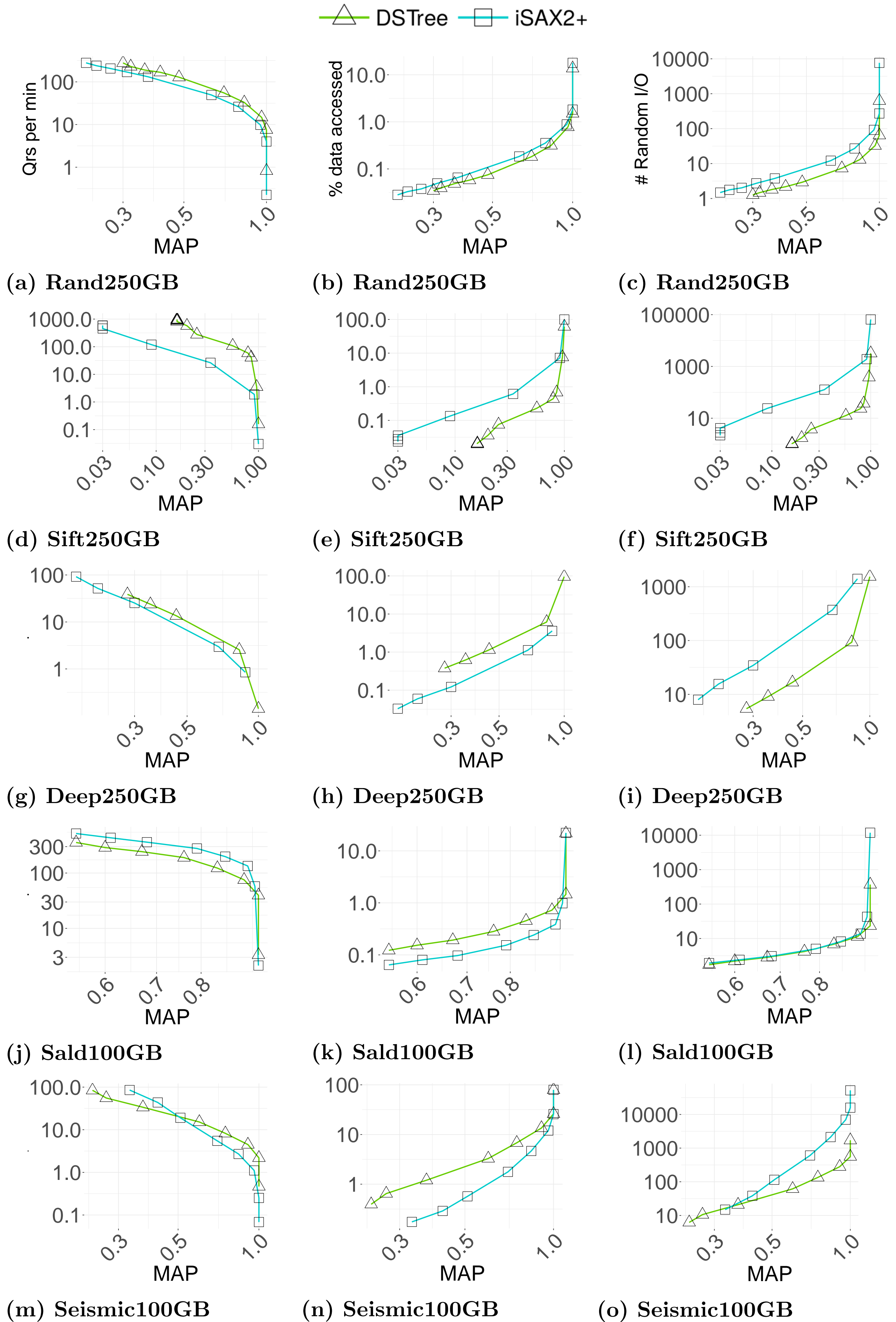


(a) **Rand250GB($\delta\epsilon$)** (b) **Rand250GB($\delta\epsilon$)** (c) **Rand250GB($\delta\epsilon$)**

(d) **Sift250GB($\delta\epsilon$)** (e) **Sift250GB($\delta\epsilon$)** (f) **Sift250GB($\delta\epsilon$)**

(g) **Deep250GB($\delta\epsilon$)** (h) **Deep250GB($\delta\epsilon$)** (i) **Deep250GB($\delta\epsilon$)**

Figure 4.6: Efficiency vs. accuracy for on disk δ-ϵ-approximate search (100NN queries)



(a) Rand250GB (b) Rand250GB (c) Rand250GB

(d) Sift250GB (e) Sift250GB (f) Sift250GB

(g) Deep250GB (h) Deep250GB (i) Deep250GB

(j) Sald100GB (k) Sald100GB (l) Sald100GB

(m) Seismic100GB (n) Seismic100GB (o) Seismic100GB

Figure 4.7: Efficiency vs. accuracy for the best methods (δ-ϵ-approximate)

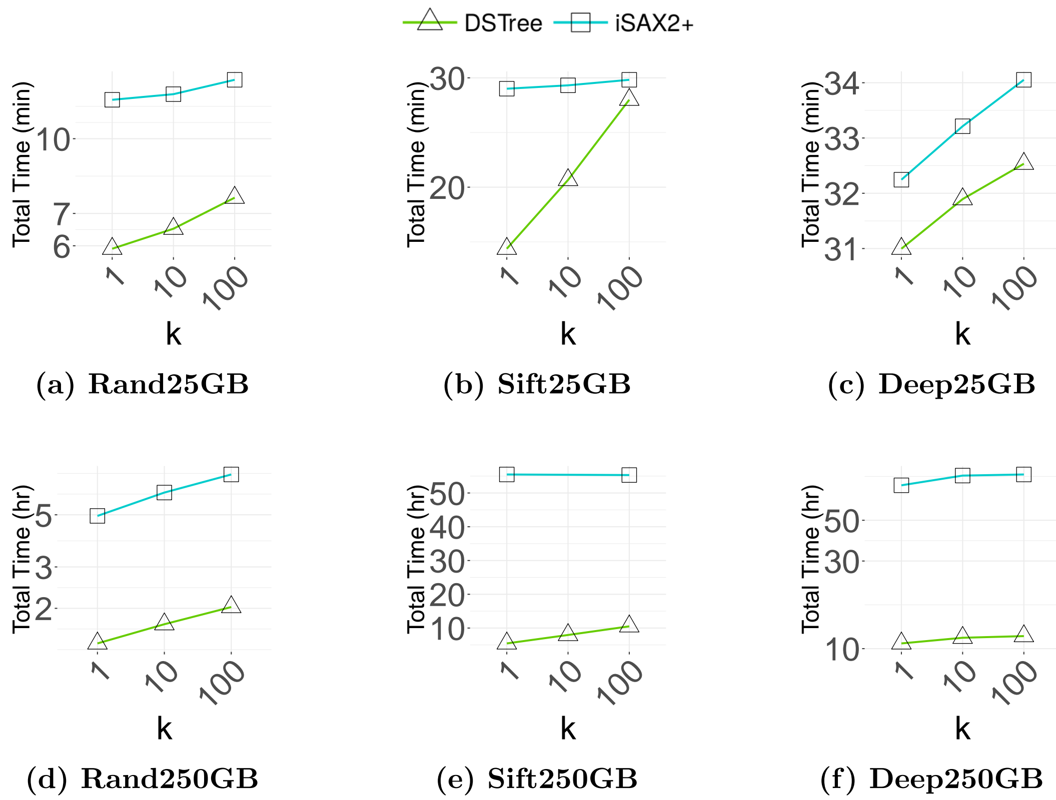
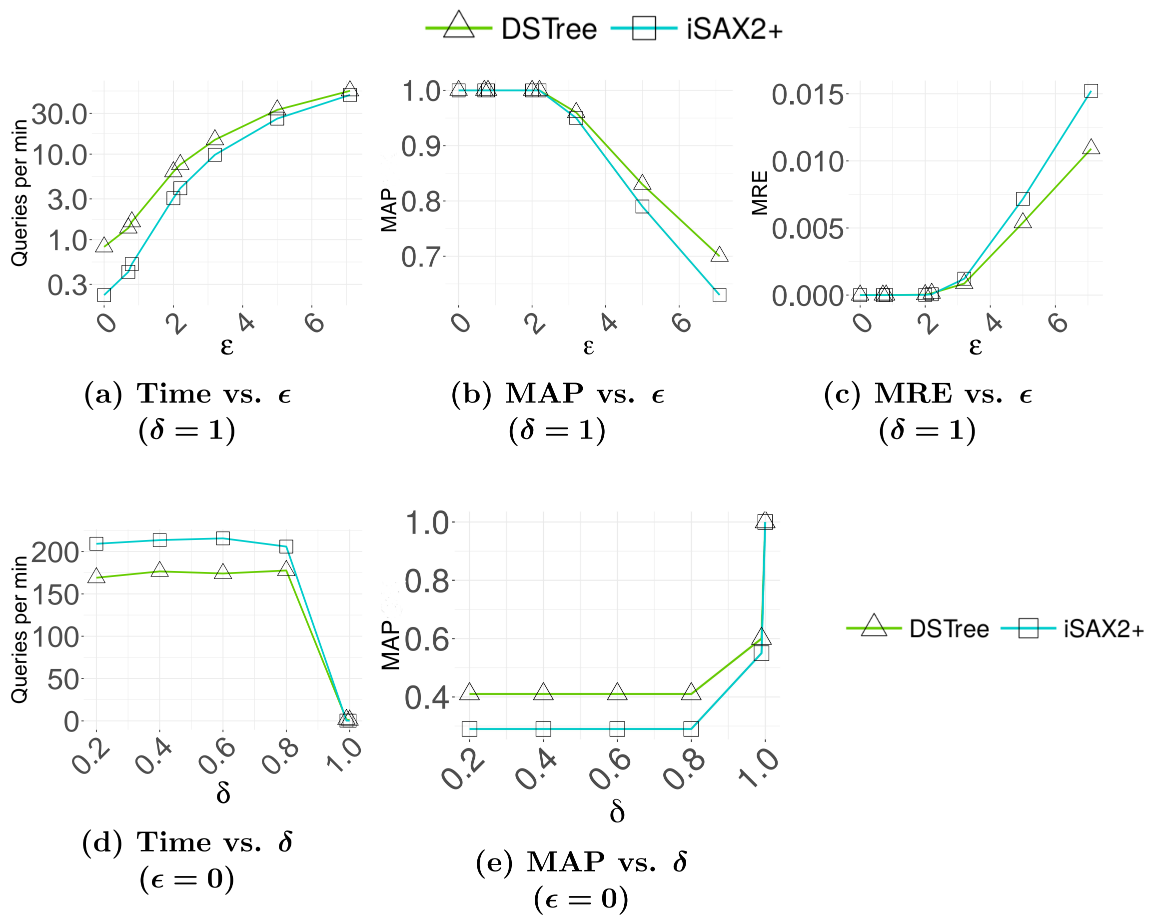


Figure 4.8: Efficiency vs. k (ϵ-approximate)



(a) Time vs. ϵ ($\delta = 1$)

(b) MAP vs. ϵ ($\delta = 1$)

(c) MRE vs. ϵ ($\delta = 1$)

(d) Time vs. δ ($\epsilon = 0$)

(e) MAP vs. δ ($\epsilon = 0$)

Figure 4.9: Accuracy and efficiency vs. δ and ϵ

4.6 Discussion

In the approximate NN search literature, experimental evaluations ignore the answering capabilities of data series methods. This is the first study that aims to fill this gap.

Unexpected Results. Some of the results are surprising:

(1) *Effectiveness of* δ*.* LSH techniques (like SRS and QALSH) exploit both δ and ϵ to tune the efficiency/accuracy tradeoff. We consider that they still fall short of expectations, because for a low ϵ, high values of δ still produce low MAP and low values of δ still result in slow execution (Figure 4.3). In the case of our extended methods, using ϵ yielded excellent empirical results, but introducing the probabilistic stop condition based on δ was ineffective (Figures 4.9-d,4.9-e). We believe that this is due to the inaccuracy of the (histogram-based) approximation of r_δ. Therefore, improving the approximation of r_δ, or devising novel approaches are interesting open research directions that will further improve the efficiency of these methods.

(2) *Approximate Query Answering with Data Series Indexes Performed Better than LSH.* Approximate query answering with DSTree and iSAX2+ outperfom SRS and QALSH (state-of-the-art LSH-based methods) both in space and time, while supporting better theoretical guarantees. This surprising result opens up exciting research opportunities, that is, devising efficient disk-based techniques that support both ng-approximate and δ-ϵ-approximate search with top performance [97]. Note that data series indexes developed for distributed platforms [44, 194] also have the potential of outperforming LSH techniques [207, 208] if extended following the ideas discussed in Section 4.3.

(3) *Our results vs. the literature.* Our results for the in-memory experiments are inline with those reported in the literature, confirming that HNSW achieves the best accuracy/efficiency tradeoff when only query answering is considered [187] (Figures 4.2a, 4.2g, 4.2j). However, when indexing time is taken into account, HNSW loses its edge to iSAX2+/DSTree for both small (Figures 4.2b, 4.2h, 4.2k) and large (Figures 4.2c, 4.2i, 4.2l) query work-

loads.

Our results for IMI show a dramatic decrease in accuracy, in terms of MAP and Avg_Recall for the Sift250GB and Deep250GB datasets, while high Avg_Recall values have been reported in the literature for the full Sift1B and Deep1B datasets [90, 35]. We thoroughly investigated the reason behind such a discrepancy and ruled out the following factors: the Z-normalization of the Sift1B/Deep1B datasets, the size of the queries, and the number of NN. We believe that our results are different for the following reasons: (a) our queries return only the number of NN requested, while the smallest candidate list considered in [90] is 10,000 for a 1-NN query; and (b) the results in [35] were obtained using training on a GPU with un-reported training sizes and times (we believe both were very large), while our focus was to evaluate methods on a CPU and account for training time. The difference in the accuracy results is most probably due to the fact that the training samples used in [35] were larger than the recommended numbers we used (1 million/4 million for the 25GB/250GB datasets, respectively). We tried to support this claim by running experiments with different training sizes: (i) we observed that increasing the training sizes for the smaller datasets improves the accuracy (the best results are reported in this study); (ii) we could not run experiments on the CPU with large training sizes for the 250GB datasets, because training was very slow: we stopped the execution after 48 hours; (iii) we tried a GPU-enabled server for training, but ran into a documented bug[5].

Practicality of QALSH, IMI and HNSW. Although QALSH provides better accuracy than SRS, it does so at a high cost: it needs to build a different index for each desired query accuracy. This is a serious drawback, while our extended methods offer a neat alternative since the index is built once and the desired accuracy is determined at query time. Although LSH methods (such as SRS) provide guarantees on the accuracy of search results, they are expensive both in time and space. The ng-approximate methods

[5] https://github.com/facebookresearch/faiss/issues/67

overcome these limitations. IMI and HNSW are considered the state-of-the-art in this category, and while they deliver better speed-accuracy tradeoffs than QALSH and SRS, they suffer from two major limitations: (a) having no guarantees can lead them to return incomplete result sets, for instance retrieving only a subset of the neighbors for a k-NN query and returning null values for the others; (b) they are very difficult to tune, which hinders their practicality. In fact, the speed-accuracy tradeoff is not determined only at query time, but also during index building, which means that an index may need to be built many times using different parameters before finding the right speed-accuracy tradeoff. This means that the optimal settings may differ across datasets, and even for different dataset sizes of the same dataset. Moreover, if the analyst builds an index with a particular accuracy target, and then their needs change, they will have to rebuild the index from scratch and go through the same process of determining the right parameter values.

For example, we built the IMI index for the Deep250GB dataset 8 times. During each run that required over 42 hours, we varied the PQ encoding sizes, the number of centroids, and training sizes but still could not achieve the desired accuracy. Regarding HNSW, we tried three different combinations of parameters (M/efConstruction = 4/500, 16/500, 48/200) for each dataset before choosing the optimal one; each run took over 40 hours on the small Deep25GB. Overall, we observe that using IMI and HNSW in practice is cumbersome and time consuming. Developing auto-tuning methods for these techniques is both an interesting problem and a necessity.

Importance of guarantees. In the approximate search literature, accuracy has been evaluated using recall, and approximation error. LSH techniques are considered the state-of-the-art in approximate search with theoretically proven sublinear time performance and probabilistic guarantees on accuracy (approximation error). Our results indicate that using the approximate search functionality of data series techniques provides tighter bounds than LSH (since δ can be equal to 1), and a much better performance in prac-

tice, with experimental accuracy levels well above the theoretical accuracy guarantees (Figure 4.9c). Note that LSH techniques can only provide probabilistic answers ($\delta < 1$), whereas our extended methods can also answer exact and ϵ-approximate queries ($\delta = 1$). A promising research direction is to improve the existing guarantees on these new methods, or establish additional ones: (1) by adding guarantees on query time performance; or (2) by developing probabilistic or deterministic guarantees on the recall or MAP value of a result set, instead of the commonly used distance approximation error. Remember that recall and MAP are better indicators of accuracy, because even small approximation errors may still result in low recall/MAP values (Figure 4.4b).

Improvement of ng-approximate methods. Our results indicate that ng-approximate query answering with exact methods offers a viable alternative to existing methods, particularly because index building is much faster and query efficiency/accuracy tradeoffs can be determined at query time. Besides, the performance of DSTree and iSAX2+ supporting ng-approximate and δ-ϵ-approximate search can be greatly improved by exploiting modern hardware (including SIMD vectorization, multi-cores, multi-sockets, and GPUs).

Incremental approximate k-NN. We established that, on some datasets, a kNN query incurs a much higher time cost as k increases. Therefore, a future research direction is to build δ-ϵ-approximate methods that support incremental search, i.e., returning the neighbors one by one as they are found. The current approaches return the k nearest neighbors all at once which impedes their interactivity.

Progressive Query Answering. The excellent empirical results with approximate search using exact methods paves the way for another very promising research direction: progressive query answering [209]. New approaches can be devised to return intermediate results with increasing accuracy until the exact answers are found.

Recommendations. Choosing the best approach to answer an approximate similarity search query depends on a variety of factors including the accuracy desired, the dataset

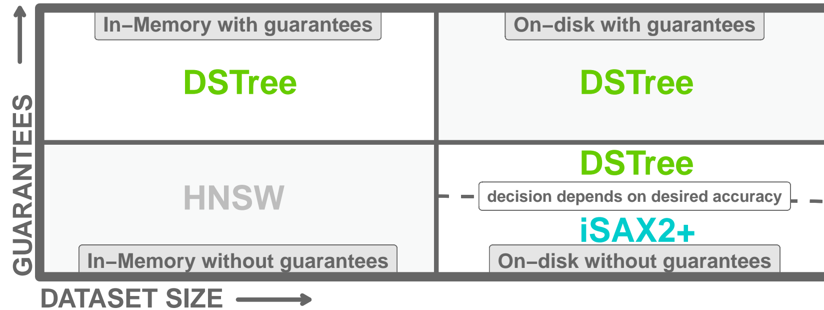


Figure 4.10: Recommendations (query answering).

characteristics, the size of the query workload, the presence of an existing index and the hardware. Figure 4.10 illustrates a decision matrix that recommends the best technique to use for answering a query workload using an existing index. Overall, DSTree is the best performer, with the exceptions of ng-approximate queries, where iSAX2+ also exhibits excellent performance, and of in-memory datasets, where HNSW is the overall winner. Accounting for index construction time as well, DSTree becomes the method of choice across the board, except for small workloads, where iSAX2+ wins.

4.7 Conclusions

In this chapter, we proposed extensions of exact data series methods that can answer δ-ϵ-approximate queries. Our proposed techniques are the clear winners for δ-ϵ-approximate similarity search for both in-memory and disk based data, while they are the only viable solution for on-disk data. These techniques are also the fastest at indexing and have the lowest footprint. The only scenario where our techniques are outperformed by another method is for in-memory ng-approximate search where HNSW [29] is the best contender. However, this kNN graph-based method has very high footprint, is difficult to tune and can return incomplete results.

We also presented the framework and results of a thorough experimental evaluation of the state-of-the-art approximate techniques from both the data series and high-dimensional vector indexing communities. Our results reveal the weaknesses and the

strengths of the different techniques sharing insights that have never been published in the literature. For instance, LSH techniques such as SRS and QALSH exploit both δ and ϵ to tune the efficiency/accuracy tradeoff. We show that their performance is still inadequate, because a low ϵ and a high δ can still lead to inaccurate answers and low values of ϵ and δ can still result in slow execution. The *ng*-approximate methods IMI and HNSW provide better efficiency but they suffer from three major limitations: (a) they have no guarantees; (b) they are very difficult to tune; and (c) their speed-accuracy tradeoff is not determined only at query time, but also during index building.

In addition, we point to unexplored promising research directions in the approximate similarity search field such as the importance of devising novel stopping criteria, establishing better guarantees and supporting progressive query answering.

Chapter 5

Hercules: A New Similarity Search Technique

Similarity search over high-dimensional objects is a critical algorithm for a variety of real-world applications. This problem has been extensively studied over the past two decades leading to a number of exact and approximate approaches. The results of two comprehensive experimental evaluations of exact and approximate similary search methods form the foundations of Hercules, a novel similarity search technique that can efficiently support exact, δ-ϵ-approximate and ng-approximate search over massive collections of high-dimensional vectors.

In this chapter[1], we describe the indexing and query answering algorithms of Hercules. Our extensive experimental study demonstrates the superiority of Hercules against the state-of-the-art approaches from the data series and high-dimensional communities.

The chapter is organized in five sections. We summarize our contributions in section 5.1, briefly survey related work in section 5.2, describe the Hercules indexing and querying algorithms in section 5.3, present results of a thorough experimental evaluation in section 5.4 and draw conclusions in section 5.6.

[1]This chapter is a slightly modified version of [99]

5.1 Main Contributions

Our main contributions are as follows:

1. Hercules leverages the insights gained from our extensive experimental studies [95, 96] about the intricate inner workings of the different similarity search methods, their strengths and weaknesses.

2. Since no single method was identified as an overall winner in exact query answering [95], Hercules incorporates key ideas from two different exact algorithms to become an overall winner in-memory and on-disk: a) efficient index clustering using DSTree's data-adaptive segmentation [25]; and b) accurate low-memory footprint summarization with SAX [156]. In addition, Hercules carefully crafts the distribution and execution of parallel instructions; exploits the Single Instruction Multiple Data (SIMD) capabilities of modern CPUs, and uses novel storage and buffering mechanisms for the index.

3. Based on the ideas we proposed [96], we extend Hercules to answer δ-ϵ-approximate and ng-approximate queries. Hercules outperforms all the state-of-the-art techniques in-memory and on-disk both in terms of accuracy and efficiency in δ-ϵ-approximate search and all disk-based techniques in ng-approximate search.

4. While Hercule's indexing is based on the DSTree's indexing algorithn, it improves its efficiency by: a) using a new storage mechanism where the data of all the index leaves is stored in one file instead of separate files physically on the disk; and b) exploiting SIMD to perform the CPU-intensive calculations required by the data-adaptive segmentation.

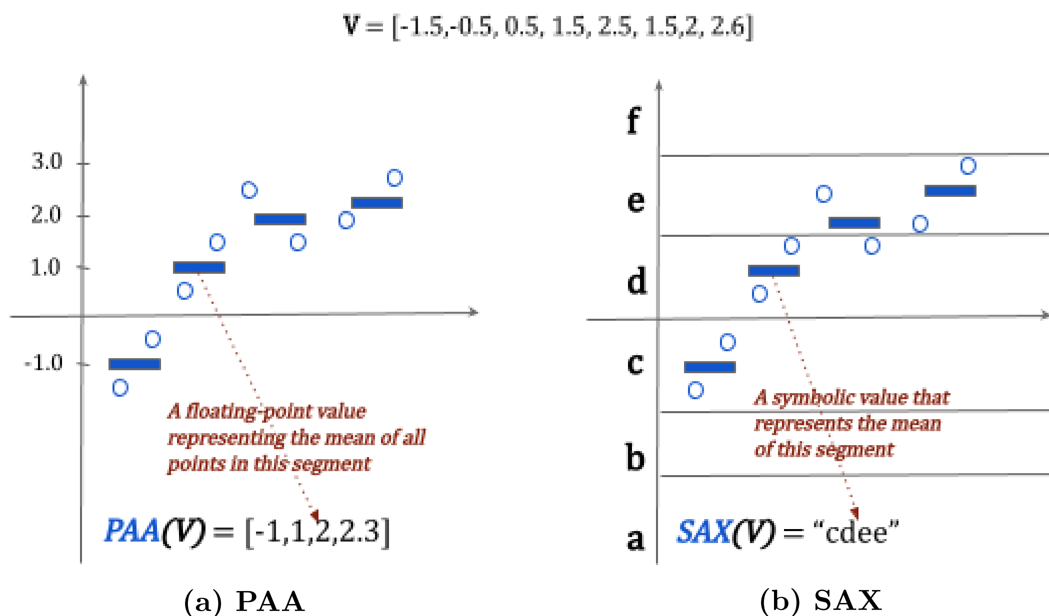
5. To evaluate Hercules, we conduct an extensive experimental evaluation using synthetic datasets and four of the largest publicly available real datasets of high-dimensional vectors, including data series, deep network embeddings and image features.

5.2 Related Work

Before presenting the workflow of Hercules, we will first give an overview of the SAX summarization and the DSTree approach.

5.2.1 SAX

The *Symbolic Aggregate Approximation* (SAX) [156] first transforms a vector V using the *Piecewise Aggregate Approximation* (PAA) [154]. The PAA summarization divides V into l equi-length segments and represents each segment with one floating-point value corresponding to the mean of all the points belonging to the segment (Fig. 5.1a). SAX reduces the footprint of the PAA representation by applying a discretization technique that map PAA values to a discrete set of symbols (alphabet) that can be succinctly represented in binary form. A SAX representation consists of l such symbols. An iSAX (indexable SAX) [157] representation can have an arbitrary alphabet size for each segment (Fig. 5.1b).

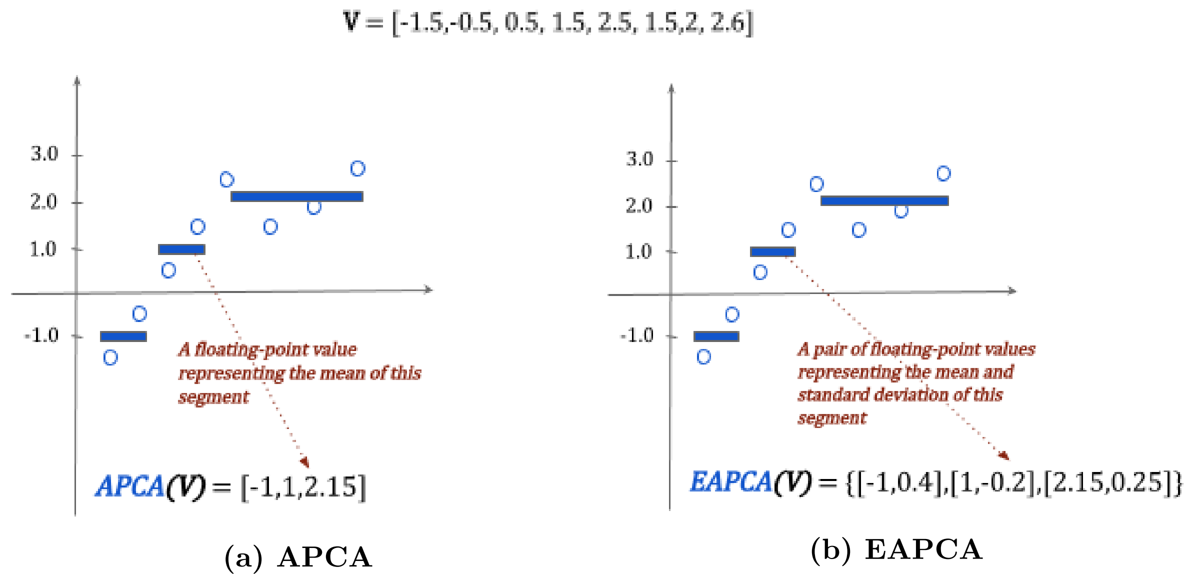


(a) PAA (b) SAX

Figure 5.1: The PAA and SAX representations

5.2.2 DSTree

The DSTree [25] approach uses the *Extended Adaptive Piecewise Approximation* (EAPCA) [25], which extends the *Adaptive Piecewise Constant Approximation* (APCA) [155] by using more information to represent each segment of a vector V. The APCA segmentation divides V into l varying-length segments and represents each segment using the mean value of the points belonging to it (Fig 5.2a). The EAPCA representation uses the standard deviation in addition to the mean to represent each segment (Fig 5.2b).



(a) APCA

(b) EAPCA

Figure 5.2: The APCA and EAPCA representations

The DSTree intertwines segmentation and indexing, building an unbalanced binary tree with two types of nodes: internal nodes and leaf nodes. Each node contains: 1) the number of vectors indexed at its subtree; 2) the segmentation $\boldsymbol{SG}$ at this node indicated by the the right endpoints of each segment; and 3) a synopsis $Z = (z_1, z_2, ..., z_m), where*$ $z_i = \{\mu_{min}, \mu_{max}, \rho_{min}, \rho_{max}\}$ that contains, for each segment, the minimum and maximum means and standard deviations of all series indexed at this node. A leaf node is associated with a filename on disk that stores all its vectors. All leaves have a maximum capacity of ψ vectors, called the leaf threshold, i.e., each leaf can store a maximum of

ψ vectors. An internal node contains pointers to the left and right children nodes and a splitting policy. Figure 5.3 shows a sample DSTree binary tree.

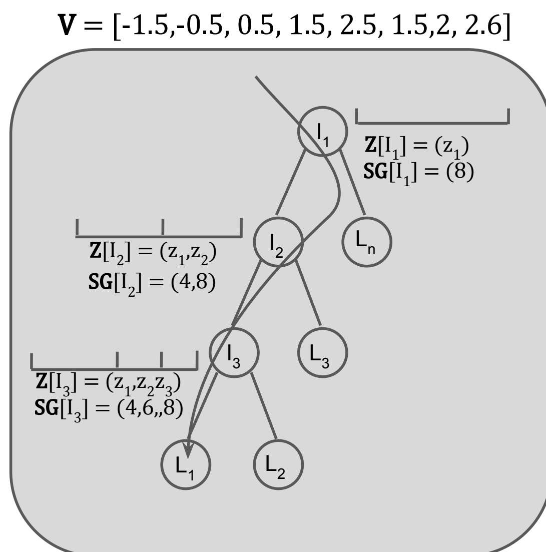


Figure 5.3: A sample binary tree index created by the DSTree

The dynamic segmentation allows the DSTree to summarize a data series more accurately than the other dimensionality reduction techniques. It is exploited in the node splitting algorithm by allowing the resolution of a summarization to increase along two dimensions: vertically and horizontally. (Instead, SAX-based indexes allow horizontal splitting by adding a breakpoint to the y-axis) In addition to a lower bounding distance, the DSTree also supports an upper bounding distance. It uses both distances to determine the optimal splitting policy for each node. The data series in a given node are all segmented using the same policy but each node has its own segmentation policy which may result in nodes having a different number of segments or segments of different lengths.

Figure 5.4 summarizes the workflow of DSTree's index building algorithm. The algorithm loads the dataset into memory one vector at a time, reading sequentially form the original file. Each loaded vector is inserted into the tree traversing the index to find the appropriate leaf, routing left or right depending on the split policy of the visited node,

and updating the synopsis of all traversed nodes. Each Leaf points to a FileBuffer which itself points to a BufferedList that contains the data stored in the leaf and currently present in-memory. Each leaf is associated with a file on disk. FileBuffers are organized in a Hash Map called the FileMap. The FileMap contains $< key, value >$ pairs where each pair is an index corresponding to a leafs Filename and the value is a pointer to the FileBuffer of the leaf.

Once memory is full, half of the FileBuffers are flushed into disk, each vector is stored in the file corresponding to the leaf where it was stored. In the final flush, The index tree and the data remaining in memory is flushed to disk.

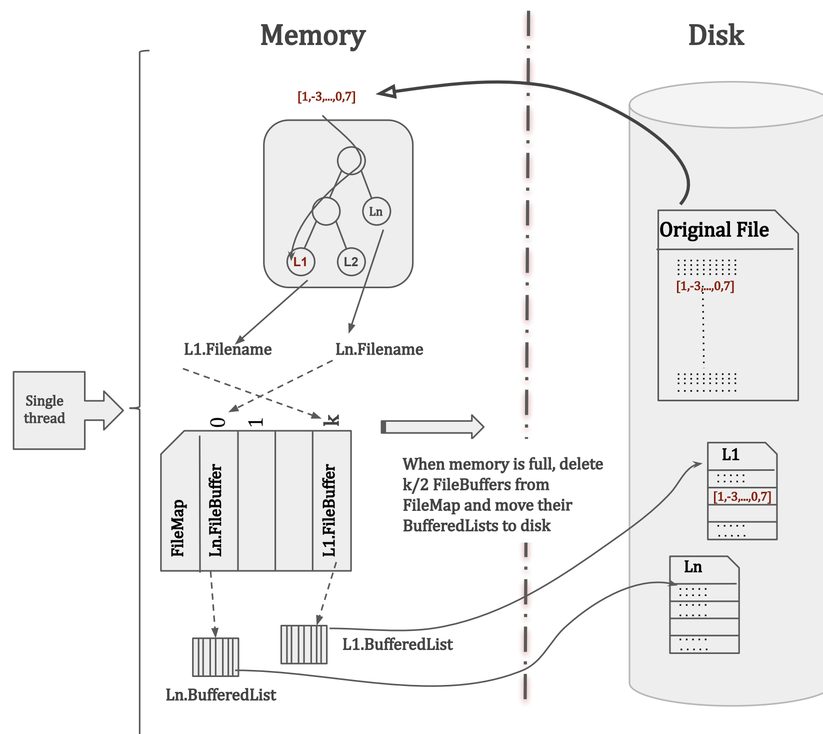


Figure 5.4: DSTree Indexing Workflow

5.3 Hercules

5.3.1 Indexing with Hercules

Hercule enhances the DSTree's indexing scheme by: a) reimplementing it from scratch in Cv and optimizing its memory management (original code was in Java); b) improving its buffering mechanism; c) using a new storage architecture where the data of all the index leaves is stored in one file instead of separate files physically on the disk; and d) exploiting multi-threading and SIMD capabilities of modern CPUs to perform the CPU-intensive calculations required by the data-adaptive segmentation.

Figure 5.5 describes the design of the Hercules indexing mechanism. The coordinator thread loads the dataset into memory by chunks, reading a small block of vectors from the original file into the CBuffer and copying vectors ready for index insertion into the WBuffer. The coordinator spaws insertWorker threads to insert vectors into the index. The insertWorker threads work in parallel to read vectors from the WBuffer and insert them into the tree. Each thread traverses the index to find the appropriate leaf, routing left or right depending on the split policy of the visited node, and updating the synopsis of all traversed nodes. The data of all inserted vectors is stored in the HardBuffer while pointers to the data belonging to each leaf are stored in the leafs SoftBuffer. When memory is full, the data in the HardBuffer is flushed to the LRD file in disk. The LRD file contains all the vectors stored in the index tree in Breadth First Search order. When all vectors have been inserted into the index tree, the tree, the LSD file and the data still in the HardBuffer are flushed to disk.

Algorithm 3 outlines Hercules' indexing algorithm. The algorithm uses a double buffering scheme to mask the CPU cost incurred building the index tree with the I/O cost involved in reading the raw vectors from disk. The coordinator thread reads the raw vectors from the original input file into the coordinator's buffer in memory, called the *CBuffer* (line 1), then copies the contents of the *CBuffer* into the *WBuffer* (line 2).

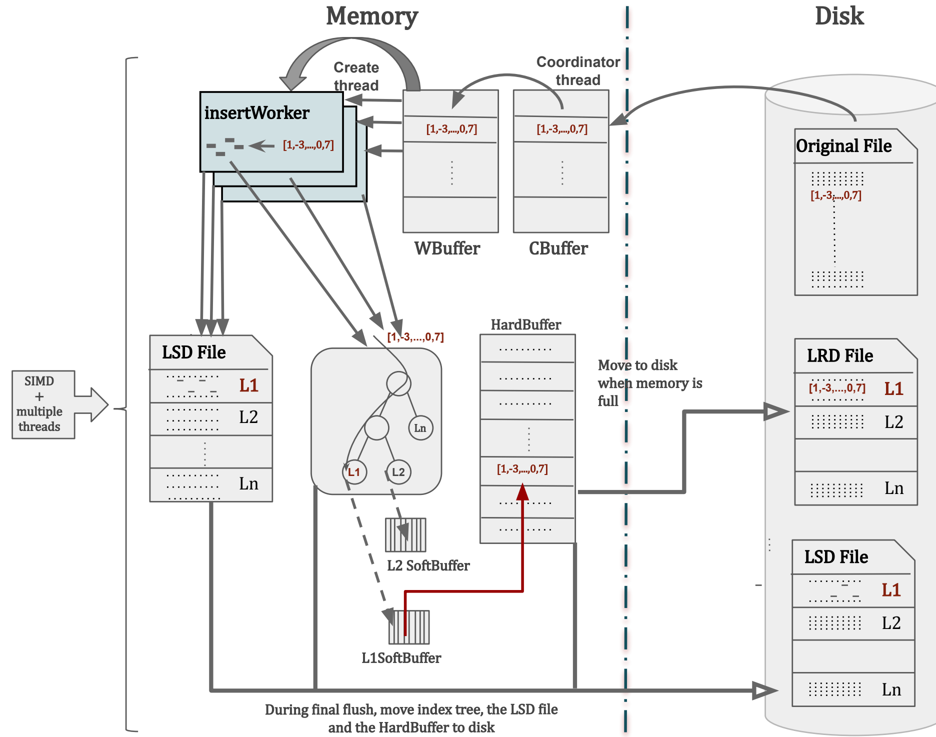


Figure 5.5: Hercules Indexing Workflow

Algorithm 3 createHerculesIndex($\boldsymbol{file}$,$\boldsymbol{idx}$,$\boldsymbol{num_threads}$)

1: read vectors from $\boldsymbol{file}$ into the $\boldsymbol{CBuffer}$;
2: copy the $\boldsymbol{CBuffer}$ contents into the $\boldsymbol{WBuffer}$;
3: **while** not EOF($\boldsymbol{file}$) **do**
4: **for** $j \leftarrow 1$ to $\boldsymbol{num_threads} - \mathbf{1}$ **do**
5: $\boldsymbol{begin} \leftarrow (j\text{-}1) * \boldsymbol{block_size}$
6: $\boldsymbol{end} \leftarrow \boldsymbol{begin} + \boldsymbol{block_size}$ - 1
7: create thread $\boldsymbol{t_j}$ to execute insertWorker($\boldsymbol{idx}$,$\boldsymbol{WBuffer}$,$\boldsymbol{begin}$,$\boldsymbol{end}$);
8: read vectors from $\boldsymbol{file}$ into the $\boldsymbol{CBuffer}$;
9: block until all insertWorker threads finished execution;
10: copy the $\boldsymbol{CBuffer}$ contents into the $\boldsymbol{WBuffer}$;
11: **if** memory is full **then**
12: **for** $\boldsymbol{j} \leftarrow 1$ to $\boldsymbol{num_threads}$ **do**
13: create thread $\boldsymbol{t_j}$ to execute flushWorker($\boldsymbol{idx}$);
14: block until all flushWorker threads finished execution;
15: flush to disk the $\boldsymbol{LSDFile}$, the $\boldsymbol{idx}$ tree and the remaining contents of the $\boldsymbol{HardBuffer}$;

Each insertWorker thread reads a block of vectors from its region in the *WBuffer*, inserts them into the index tree, calculates their SAX summaries and adds them to the LSD file.

No coordination is needed for the workers to read data from the *WBuffer*; however the insertion of vectors into the tree requires careful synchronization, which we will detail in the next paragraph. While the worker threads are inserting the vectors into the index tree, the coordinator reads another chunk of vectors from the original file (line 8) into the *CBuffer*. Once the worker threads have finished processing the vectors in the *WBuffer* (line 9), the coordinator copies the contents of the *CBuffer* into the *WBuffer* to feed the workers more vectors for insertion (line 10). If the algorithm runs out of memory, the coordinator and worker threads coordinate the flushing of the raw data into the LRD file (lines 11-13). Once the flushing has completed and provided there remains vectors to process, the worker and coordinator threads resume the same process we described at line 3. When indexing is complete, the coordinator flushes the index tree, the LSD file and any remaining vectors in the HardBuffer to disk. As we will explain in more detail later, the HardBuffer is used to hold all the vectors that have been inserted into the tree.

Each worker thread calls Algorithm 4 to insert a chunk of vectors into the index tree, calculate their SAX summaries and store them into the LSDfile. Algorithm 5 describes the steps taken by each thread to insert one vector into the index and the hand-over-hand synchronization mechanism to allow efficient and correct index building. If the tree contains only the root (line 1), a lock is acquired on the root (line 2). If the tree contains internal nodes (line 3), a lock is acquired on each traversed node to update its synopsis, unlocking it only when the appropriate child is locked and this child is not a leaf. The parent of a leaf is not unlocked to prevent another thread from being in the lock queue of this leaf while the current thread is processing it, particularly if the leaf needs to be split and becomes an internal node. Once a leaf is locked, its synopsis is updated with the vector information (line 10) then the vector is appended to it (line 11).

Note that a vector is appended to a node by having the leaf node point to a SoftBuffer which itself contains pointers to the actual data in the HardBuffer. We opted for this architecture because it exhibited the best performance. In fact, we found that allocating

a large memory buffer at the start of the index creation and releasing it once all vectors have been inserted is more efficient than having each leaf continuously allocate and release its own memory buffer because issuing a smaller number of system calls: 1) results in a faster performance; 2) reduces the occurrence of out-of-memory management issues (when a program issues a large number of memory cleanup operations, the memory can be retained by the process for later reuse, so it is not purged, i.e., it is not returned to the operating system which considers the memory to still be in use and can terminate the process for an out-of-memory error [210]).

If a leaf node reaches its maximum capacity (line 12), i.e. ψ, the best splitting policy is determined following the same heuristics as in the DSTree [25] (line 13) and the node is split into two children nodes (line 14). The vectors in the split node are redistributed among the left and right children nodes according to the node's splitting policy (Algorithm 7). Once the vector is inserted, the locks on the leaf and its parent node (except when the leaf is the root) are released (lines 20-22).

Algorithm 4 insertWorker($\boldsymbol{idx}$,$\boldsymbol{B}$,$\boldsymbol{begin}$,$\boldsymbol{end}$)

1: $\boldsymbol{root} \leftarrow$ root of $\boldsymbol{idx}$;
2: **for** $\boldsymbol{i} \leftarrow \boldsymbol{begin}$ to $\boldsymbol{end}$ **do**
3: $\quad \boldsymbol{V_i} \leftarrow \boldsymbol{B}[\boldsymbol{i}]$;
4: $\quad$ insertVectorToNode($\boldsymbol{idx}$,$\boldsymbol{root}$,$\boldsymbol{V_i}$);
5: $\quad$ calculate SAX summary of $\boldsymbol{V_i}$ and store it in the $\boldsymbol{LSDFile}$;

Algorithm 6 updates the synopsis of a node, if necessary, based on the new vector being inserted. Recall that a node has a horizontal and a vertical segmentation and that the synopsis of each segmentation consists of the minimum and maximum values of the mean and standard deviations of each segment.

Algorithm 7 describes how Hercules routes a vector during the tree traversal. It first gets s_b and s_e, the beginning and end points of the segment that was used during the split of the traversed node (lines 1-3). Then, it calculates the mean and standard deviation of the points belonging to the segment $[s_b, s_e]$ of the vector being inserted. It picks either

Algorithm 5 insertVectorToNode(***idx***,***node***,***vector***)

1: **if** ***node*** is a leaf **then**
2: acquire lock on ***node***
3: **while** ***node*** is not a leaf **do**
4: acquire lock on ***node***
5: updateNodeSynopsis(***node, vector***);
6: ***node*** ← routeToChild(***node, vector***)
7: acquire lock on ***node***
8: **if** ***node*** is not a leaf **then**
9: release lock on ***node*** parent
10: updateNodeSynopsis(***node, vector***);
11: appendVectorToNode(***node, vector***);
12: **if** ***node*** is full **then**
13: ***policy*** ← *getBestSplitPolicy*(***node***);
14: split ***node*** into two child nodes according to ***policy***;
15: get all vectors in ***node*** from memory and disk (if flushed) ;
16: **for** each vector ***V*** in ***node*** **do**
17: ***N*** ← routeToChild(***node, V***)
18: updateNodeSynopsis(***N, V***);
19: appendVectorToNode(***N, V***);
20: release lock on ***node***
21: **if** ***node*** parent is not NULL **then**
22: release lock on ***node*** parent

Algorithm 6 updateNodeSynopsis(***node***,***vector***)

1: $\boldsymbol{SG_v}$ ← the vertical segments of ***node***;
2: $\boldsymbol{SG_h}$ ← the horizontal segments of ***node***;
3: **for** each segment ***S*** in $\boldsymbol{SG_v}$ and $\boldsymbol{SG_h}$ **do**
4: ***seriesSketch*** ← calcMeanSD($\boldsymbol{V}$,$\boldsymbol{S_{start}}$,$\boldsymbol{S_{end}}$);
5: update the min/max mean and sd of ***S*** with ***seriesSketch.Mean*** and ***seriesSketch.SD***;

the mean or the standard deviation depending on the split policy of the node to compare to the threshold decided during the split (lines 6-8). The vector is routed left if the value is less than the threshold and right otherwise (lines 9-12).

Algorithm 7 routeToChild(***node***,***vector***)

1: Get the ***splitPolicy*** of ***node***;
2: ***from*** ← ***splitPolicy.segmentStart***; ▷ segment used in split
3: ***to*** ← ***splitPolicy.segmentEnd***;
4: ***seriesSketch*** ← calcMeanSD(***vector***,***from***,***to***);
5: **if** ***splitPolicy.type*** is mean **then**
6:     ***value*** ← ***seriesSketch.mean***
7: **else**
8:     ***value*** ← ***seriesSketch.sd***
9: **if** ***value*** < ***splitPolicy.value*** **then**
10:     **return** ***node.leftChild***
11: **else**
12:     **return** ***node.rightChild***

5.3.2 Query Answering with Hercules

Exact Search

An overview

Figure 5.6 describes the workflow of Hercule's query answering.

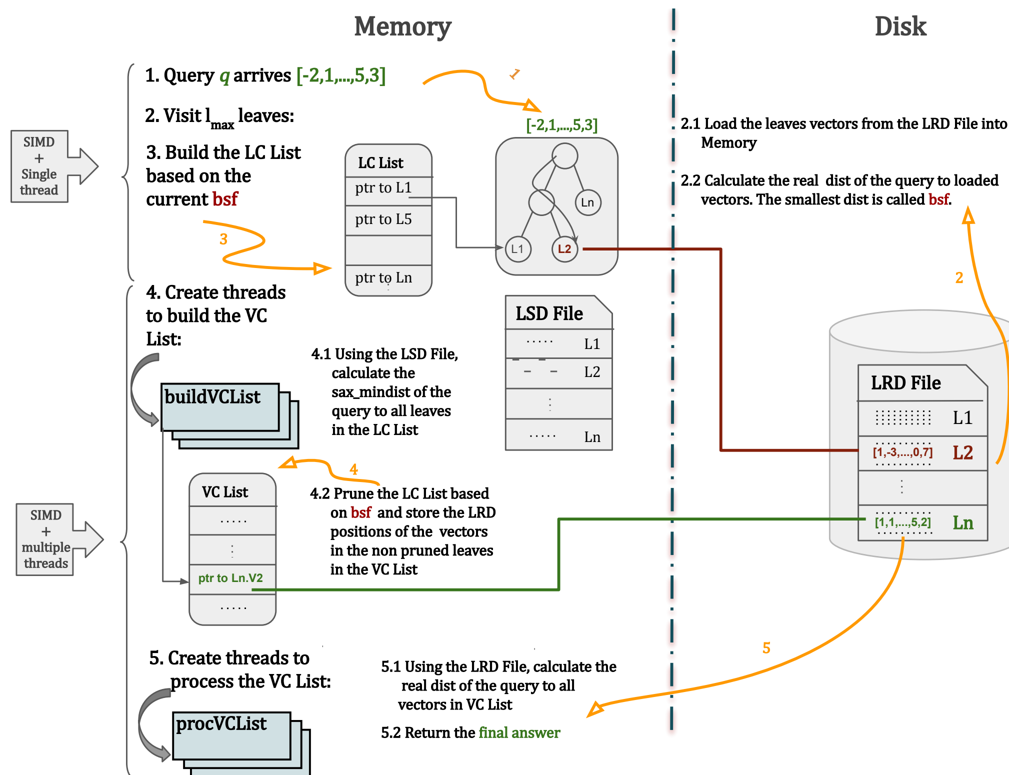


Figure 5.6: Query Answering Workflow

Algorithm 8 outlines the kNN exact nearest neighbor search with Hercules. It takes as arguments, the query vector V_Q, the index idx, the maximum number of leaves l_{max} that the approximate search can visit, the number of neighbors k, an array $kresults$ to store the k neighbors, and the number of threads that can be exploited by the algorithm. It first starts by initializing an array $results$ with the current k best-so-far answers (lines 1-2), the $LCList$ and $VCList$ arrays which will hold the candidate leaves and vectors respectively (line 3), and a priority pqueue $pqueue$ with the root node of the index (lines 4-6), where the priority is based on the EAPCAMinDist, i.e., the lower bounding distance of the query to the EAPCA segmentation of a given node as presented in [25]. Then an approximate kNN search is performed by calling the function $approxKNNSearch$, which returns k approximate neighbors for query V_Q by visiting at most l_{max} leaves. The variable $kthbsf$ is initialized with the real distance of the kth neighbor to V_Q (line 8) and is used by the $buildLCList$ function to prune the search space. The non-pruned leaves are stored in the $LCList$ (line 9). The $buildVCList$ function applies a double filter on the sax representation of the vectors belonging to the leaves in the $LCList$ using the SAXMinDist [146] and stores the non-pruned candidate vectors in the $VCList$ (line 10). The $refineVCList$ function (line 11) loads the vectors in the $VCList$ from disk and calculates their real distance to the query returning the k vectors with the minimum real Euclidean distance to V_Q as the final answers. Note that we use SIMD for efficient real distance calculations.

In what follows, we will describe in more detail the different building blocks of Algorithm 8.

Algorithm 8 exactKNNSearch($\boldsymbol{V_Q}$, $\boldsymbol{idx}$, $\boldsymbol{l_{max}}$, $\boldsymbol{k}$, $\boldsymbol{kresults}$, $\boldsymbol{numThreads}$)

1: **for** each $\boldsymbol{result}$ in $\boldsymbol{kresults}$ **do**
2: $\boldsymbol{result.dist} \leftarrow \infty$; $\boldsymbol{result.pos} \leftarrow NULL$;
3: initialize the $\boldsymbol{LCList}$ and $\boldsymbol{VCList}$ arrays;
4: initialize a priority queue $\boldsymbol{pqueue}$
5: $\boldsymbol{rootMinDist} \leftarrow$ calcEAPCAMinDistSIMD($\boldsymbol{V_Q}$, $\boldsymbol{rootNode}$);
6: add $\boldsymbol{rootNode}$ to $\boldsymbol{pqueue}$ with priority $\boldsymbol{rootMinDist}$;
7: approxKNNSearch($\boldsymbol{V_Q}$, $\boldsymbol{LRDFile}$, $\boldsymbol{pqueue}$, $\boldsymbol{l_{max}}$, $\boldsymbol{k}$, $\boldsymbol{kresults}$);
8: $\boldsymbol{kthbsf} \leftarrow \boldsymbol{kresults[k-1]}$;
9: buildLCList($\boldsymbol{V_Q}$, $\boldsymbol{LRDFile}$, $\boldsymbol{kthbsf}$, $\boldsymbol{pqueue}$, $\boldsymbol{LCList}$);
10: buildVCList($\boldsymbol{V_Q}$, $\boldsymbol{LSDFile}$, $\boldsymbol{kthbsf}$, $\boldsymbol{LCList}$, $\boldsymbol{VCList}$, $\boldsymbol{numThreads}$);
11: refineVCList($\boldsymbol{V_Q}$, $\boldsymbol{LRDFile}$, $\boldsymbol{k}$, $\boldsymbol{kresults}$, $\boldsymbol{VCList}$, $\boldsymbol{numThreads}$);

Finding the first bsf

The kth approximate answer, called $kthbsf$ will serve to prune the search space. Algorithm 9 finds k approximate first baseline answers and stores them in the array $kresults$ in increasing order of the real Euclidean distance. It visits a maximum of l_{max} leaves, where l_{max} is a parameter provided by the user and is 1 by default. In line 2, it pops the element in $pqueue$ with the highest priority, i.e., the node with the lowest EAPCAMinDist to the query. To understand the reason behind allocating a higher priority to the EAPCAMinDist, recall that it is a lower-bounding distance. Therefore, if a popped node has an EAPCAMinDist value greater than the current $kthbsf$ answer, this means that the search is complete (line 5) since any vector in the subtree rooted at this node has a real distance that is greater than or equal to EAPAMinDist and thus cannot improve the $kthbsf$. Besides, since the priority in $pqueue$ is based on the minimum EAPCAMinDist, all remaining nodes in $pqueue$ can be pruned because their lower-bounding distances will also be greater than the $kthbsf$. Note that we use SIMD operations to speed up the calculations of the EAPCAMinDist.

If the non-pruned node is a leaf (line 6), then the vectors of this leaf are read from the LRDFile (line 7) and their real Euclidean distances to the query are calculated (line 10) and the $kreults$ array is updated if applicable (lines 11-15). The algorithm stops improving the k answers once the leaves threshold is reached (line 16).

Algorithm 9 approxKNNSearch($\boldsymbol{V_Q}$, $\boldsymbol{LRDFile}$, $\boldsymbol{pqueue}$, $\boldsymbol{l_{max}}$, $\boldsymbol{k}$, $\boldsymbol{kresults}$)

```
 1: visitedLeaves ← 0
 2: while (qelement ← pop next element from pqueue) do
 3:     n ← qelement.node;
 4:     kthbsf ← kresults[k − 1];
 5:     if n.dist > kthbsf.dist then break;
 6:     if n is a leaf then                                   ▷ a leaf node
 7:         read vectors of n from the LRDFile;
 8:         for each V_C in n do
 9:             kthbsf ← kresults[k − 1];
10:             realDist ← calcRealDistSIMD(V_Q,V_C);
11:             if realDist < kthbsf.dist then
12:                 create new result called bsf;
13:                 bsf.dist ← realDist ;
14:                 bsf.pos ← position of V_C in LRDFile;
15:                 add bsf to kresults;
16:         if visitedLeaves > l_max then break;
17:     else                                                  ▷ an internal node
18:         for each childNode in n do
19:             kthbsf ← kresults[k − 1];
20:             minDist ← calcEAPCAMinDistSIMD(S_Q,childNode);
21:             if minDist < kthbsf.dist then add childNode to
22:                 pqueue with priority minDist;
```

Instead, if the non-pruned node is an internal node (line 17), its children are added to *pqueue* if their EAPCAMinDist is smaller than the current $kthbsf$ (line 18-22). Then the algorithm resumes processing the *pqueue* unless it is empty (line 2) and returns the approximate k answers to the exact algorithm to help prune the search space.

Creating the LCList

Once k approximate answers have been found, the kth approximate answer, called $kthbsf$ is used by Algorithm 10 to prune the search space and create the LCList. In addition to the $kthbsf$, the algorithm takes as arguments, the query vector V_Q, the index idx, the priority queue *pqueue*, and an array $LCList$ to store the non-pruned leaves.

In algorithm 10, the search in the index tree resumes with the remaining nodes in *pqueue* (line 2), i.e., nodes that were visited by algorithm 9 are not accessed again. If the node's EAPCAMinDist is larger than the $kthbsf$ distance, the algorithm stops (line 4).

Algorithm 10 buildLCList($\boldsymbol{V_Q}$, $\boldsymbol{LRDFile}$, $\boldsymbol{kthbsf}$, $\boldsymbol{pqueue}$, $\boldsymbol{LCList}$)

```
1: count ← 0;
2: while (qelement ← pop next element from pqueue) do
3:     n ← qelement.node;
4:     if n.dist > kthbsf.dist then break;
5:     if n is a leaf then                                        ▷ a leaf node
6:         LCList[count].node ← n;
7:         LCList[count].dist ← qelement.dist;
8:         count ← count + 1;
9:     else                                                    ▷ an internal node
10:        for each childNode in n do
11:            minDist ← calcEAPCAMinDistSIMD(V_Q,childNode);
12:            if minDist < kthbsf.dist then add childNode to
13:                pqueue with priority minDist;
14: sort the candidate leaves in the LClist in increasing order of their position in the
    LRDFile;
```

Otherwise it adds non-pruned leaves into the LCList (lines 5-8) and non-pruned internal nodes into *pqueue* (lines 9-13). Note that leaves are treated differently in Algorithms 9 and 10 because in the former, the vectors of the leaves are loaded from disk and the real distance is calculated between each vector and the query, updating *kresults* as necessary, whereas in the latter, the *kresults* are not updated and pointers to the leaves are stored for further processing, so the disk is not accessed in this case.

Once all nodes in *pqueue* have been processed, the candidate leaves in the *LCList* are sorted in increasing order of their position in the *LRDFile* (line 14). This is to reduce the overhead of disk random I/O by ensuring that data pages are visited in the order they are laid out on disk.

Creating the VCList

While the previous building blocks of the exact search algorithm use SIMD to efficiently calculate the distances EAPCAMinDist and RealDist, they run sequentially. We now describe Algorithm 11, the multi-threaded algorithm that creates the *VCList*. It starts initializing the threads (line 1) and their local data. Each thread maintains a local *VCList* (line 2) of size *count* (line 3) and current index *currentCandidate* (line 4) and

Algorithm 11 buildVCList($\boldsymbol{V_Q}$, $\boldsymbol{LSDFile}$, $\boldsymbol{kthbsf}$, $\boldsymbol{LCList}$, $\boldsymbol{VCList}$, $\boldsymbol{numThreads}$)

1: initialize threads;
2: each thread maintains a local $\boldsymbol{VCList}$;
3: threads $\boldsymbol{count} \leftarrow 0$;
4: threads $\boldsymbol{currentCandidate} \leftarrow 0$;
5: **for** $\boldsymbol{i} \leftarrow 1$ to $\boldsymbol{numThreads}$ **do**
6: create thread $\boldsymbol{t_j}$ to execute buildVCListWorker($\boldsymbol{V_Q}$, $\boldsymbol{LSDFile}$, $\boldsymbol{kthbsf}$, $\boldsymbol{LCList}$, $\boldsymbol{thread.VCList}$);
7: merge the threads local $\boldsymbol{VCList}$ into a global $\boldsymbol{VCList}$;

Algorithm 12 buildVCListWorker($\boldsymbol{V_Q}$, $\boldsymbol{LSDFile}$, $\boldsymbol{kthbsf}$, $\boldsymbol{LCList}$, $\boldsymbol{VCList}$)

1: **while** $\boldsymbol{currentCandidate} < \boldsymbol{LCList.size}$ **do**
2: n $\leftarrow$ $\boldsymbol{LCList}$[fetchAndAdd($\boldsymbol{currentCandidate}$)];
3: **for** each $\boldsymbol{S_C}$ in $\boldsymbol{n}$ **do**
4: read SAX summary $\boldsymbol{S_C}$ of each vector $\boldsymbol{V_C}$ of leaf $\boldsymbol{n}$ from the $\boldsymbol{LSDFile}$;
5: $\boldsymbol{minDist} \leftarrow$ calcSAXMinDist($\boldsymbol{PAA(V_Q)}$,$\boldsymbol{S_C}$);
6: **if** $\boldsymbol{minDist} < \boldsymbol{kthbsf.dist}$ **then**
7: add $\boldsymbol{V_C}$ to the thread's local $\boldsymbol{VCList}$;

executes a $buildVCListWorker$ (line 6) to populate its $VCList$. Once all threads have terminated execution, their local lists are merged into a global $VCList$ (line 7).

Algorithm 12 outlines the tasks of each $buildVCListWorker$. Each worker processes a leaf from the $LCList$ if it is not empty (line 1) using a fetch-and-add operation for concurrency control (line 2). For every vector S_C in the leaf, it reads its SAX summary from the $LSDFile$ (line 4), which is in-memory, calculates its SAXMinDist [146] to the query (line 5) and inserts its $LRDFile$ position into the thread's $VCList$ if the vector cannot be pruned (lines 6-7).

Refining the VCList

Algorithm 13 and describes the last step in exact search which involves refining the vectors in the $VCList$ to return the exact k neighbors of the query vector V_Q. Each thread keep track of the variable $currentCandidate$ to fetch the correct vector from the $VCList$ and spawns a worker that processes vectors from the $VCList$ in algorithm 14.

As long as there are unprocessed elements in the $VCList$ (line 1), the worker in

Algorithm 13 refineVCList($\boldsymbol{V_Q}$, $\boldsymbol{LRDFile}$, $\boldsymbol{k}$, $\boldsymbol{kresults}$, $\boldsymbol{VCList}$, $\boldsymbol{numThreads}$)

1: initialize threads;
2: threads $\boldsymbol{currentCandidate} \leftarrow 0$;
3: **for** $\boldsymbol{i} \leftarrow 1$ to $\boldsymbol{numThreads}$ **do**
4: create thread $\boldsymbol{t_j}$ to execute refineVCListWorker($\boldsymbol{V_Q}$,$\boldsymbol{LRDFile}$,$\boldsymbol{k}$,$\boldsymbol{kresults}$,$\boldsymbol{VCList}$);

Algorithm 14 refineVCListWorker($\boldsymbol{V_Q}$, $\boldsymbol{LRDFile}$, $\boldsymbol{k}$, $\boldsymbol{kresults}$, $\boldsymbol{VCList}$)

1: **while** $\boldsymbol{currentCandidate} < \boldsymbol{VCList.size}$ **do**
2: $\boldsymbol{kthbsf} \leftarrow \boldsymbol{kresults}[\boldsymbol{k-1}]$;
3: $\boldsymbol{ptr} \leftarrow \boldsymbol{VCList}$[fetchAndAdd($\boldsymbol{currentCandidate}$)];
4: read the $\boldsymbol{V_C}$ with location $\boldsymbol{ptr}$ from the $\boldsymbol{LRDFile}$;
5: $\boldsymbol{realDist} \leftarrow$ calcRealDistSIMD($\boldsymbol{V_Q}$,$\boldsymbol{V_C}$);
6: **if** $\boldsymbol{realDist} < \boldsymbol{kthbsf.dist}$ **then**
7: create new result called $\boldsymbol{bsf}$;
8: $\boldsymbol{bsf.dist} \leftarrow \boldsymbol{realDist}$;
9: $\boldsymbol{bsf.pos} \leftarrow \boldsymbol{ptr}$;
10: atomically add $\boldsymbol{bsf}$ to $\boldsymbol{kresults}$;

algorithm 14 loads vectors from the $VCList$ from disk (line 4) and calculates their real distance to the query using efficient SIMD calculations (line 5) and atomically updating the $kresults$ array as needed (lines 6-10).

When all threads spawned by algorithm 13 have finished execution the $kresults$ array will contain the k vectors with the minimum real Euclidean distance to V_Q.

ng-Approximate Search

The ng-approximate search of Hercules is Algorithm 9. The accuracy/efficiency trade-off is determined using different values for l_{max}; visiting more leaves improves accuracy at the expense of speed.

δ-ϵ-Approximate Search

We established in [96] how an index-based exact similarity search algorithm can be modified to support δ-ϵ-approximate search. Following these ideas, we propose Algorithm 15 which supports δ-ϵ-approximate search over a Hercules index.

Algorithm 15 DeltaEpsilonKNNSearch($\boldsymbol{V_Q}$, $\boldsymbol{idx}$, $\boldsymbol{l_{max}}$, $\boldsymbol{k}$, $\boldsymbol{kresults}$, $\boldsymbol{numThreads}$, $\boldsymbol{\delta}$, $\boldsymbol{\epsilon}$, $\boldsymbol{F_Q(.)}$)

1: **for** each $\boldsymbol{result}$ in $\boldsymbol{kresults}$ **do**
2: $\boldsymbol{result.dist} \leftarrow \infty$; $\boldsymbol{result.pos} \leftarrow NULL$;
3: initialize the $\boldsymbol{LCList}$ and $\boldsymbol{VCList}$ arrays;
4: initialize a priority queue $\boldsymbol{pqueue}$
5: $\boldsymbol{rootMinDist} \leftarrow$ calcEAPCAMinDistSIMD($\boldsymbol{V_Q}$, $\boldsymbol{rootNode}$);
6: add $\boldsymbol{rootNode}$ to $\boldsymbol{pqueue}$ with priority $\boldsymbol{rootMinDist}$;
7: approxKNNSearch($\boldsymbol{V_Q}$, $\boldsymbol{LRDFile}$, $\boldsymbol{pqueue}$, $\boldsymbol{l_{max}}$, $\boldsymbol{k}$, $\boldsymbol{kresults}$);
8: $\boldsymbol{kthbsf} \leftarrow \boldsymbol{kresults}[\boldsymbol{k-1}]$;
9: buildLCListEpsilon($\boldsymbol{V_Q}$, $\boldsymbol{LRDFile}$, $\boldsymbol{kthbsf}$, $\boldsymbol{pqueue}$, $\boldsymbol{LCList}$, $\boldsymbol{\epsilon}$);
10: buildVCList($\boldsymbol{V_Q}$, $\boldsymbol{LSDFile}$, $\boldsymbol{kthbsf}$, $\boldsymbol{LCList}$, $\boldsymbol{VCList}$, $\boldsymbol{numThreads}$);
11: $\boldsymbol{r_\delta(Q)} \leftarrow$ calcDeltaRadius($\boldsymbol{S_Q}$,$\boldsymbol{\delta}$, $\boldsymbol{F_Q(.)}$);
12: refineVCListDeltaEpsilon($\boldsymbol{V_Q}$, $\boldsymbol{LRDFile}$, $\boldsymbol{k}$, $\boldsymbol{kresults}$, $\boldsymbol{VCList}$, $\boldsymbol{numThreads}$,$\boldsymbol{r_\delta(Q)}$,$\boldsymbol{\epsilon}$);

Algorithm 16 modifies lines 4 and 12 of Algorithm 10 and Algorithm 16 modifies line 11 of Algorithm 14. Details on how to calculate $r_\delta(Q)$ can be found in [18, 198] for $k = 1$ and $k \geq 1$, respectively.

Algorithm 16 buildLCListEpsilon($\boldsymbol{V_Q}$, $\boldsymbol{LRDFile}$, $\boldsymbol{kthbsf}$, $\boldsymbol{pqueue}$, $\boldsymbol{LCList}$, $\boldsymbol{\epsilon}$)

```
1:  count ← 0;
2:  while (qelement ← pop next element from pqueue) do
3:      n ← qelement.node;
4:      if n.dist > kthbsf.dist/(1 + ε) then break;
5:      if n is a leaf then                                   ▷ a leaf node
6:          LCList[count].node ← n;
7:          LCList[count].dist ← qelement.dist;
8:          count ← count + 1;
9:      else                                                  ▷ an internal node
10:         for each childNode in n do
11:             minDist ← calcEAPCAMinDistSIMD(V_Q,childNode);
12:             if minDist < kthbsf.dist/(1 + ε) then add childNode to
13:                 pqueue with priority minDist;
14: sort the candidate leaves in the LClist in increasing order of their position in the
    LRDFile;
```

Algorithm 17 refineVCListDeltaEpsilon($\boldsymbol{V_Q}$, $\boldsymbol{LRDFile}$, $\boldsymbol{k}$, $\boldsymbol{kresults}$, $\boldsymbol{VCList}$, $\boldsymbol{numThreads}$, $\boldsymbol{r_\delta(Q)}$, $\boldsymbol{\epsilon}$)

```
1: initialize threads;
2: threads currentCandidate ← 0;
3: for i ← 1 to numThreads do
4:     create thread t_j to execute
        refineVCListWorkerDeltaEpsilon(V_Q,LRDFile,k,kresults,VCList, r_δ(Q), ε);
```

Algorithm 18 refineVCListWorkerDeltaEpsilon($\boldsymbol{V_Q}$, $\boldsymbol{LRDFile}$, $\boldsymbol{k}$, $\boldsymbol{kresults}$, $\boldsymbol{VCList}$, $\boldsymbol{r_\delta(Q)}$, $\boldsymbol{\epsilon}$)

```
1:  while currentCandidate < VCList.size do
2:      kthbsf ← kresults[k − 1];
3:      ptr ← VCList[fetchAndAdd(currentCandidate)];
4:      read the V_C with location ptr from the LRDFile;
5:      realDist ← calcRealDistSIMD(V_Q,V_C);
6:      if realDist < kthbsf.dist then
7:          create new result called bsf;
8:          bsf.dist ← realDist;
9:          bsf.pos ← ptr;
10:         atomically add bsf to kresults;
11:         if bsf.dist ≤ (1 + ε) r_δ(Q) then exit;
```

5.3.3 Complexity Analysis

In the following paragraphs, we analyze the space complexity for the Hercules indexing structure, as well as the time complexity for its exact and approximate search algorithms. Since the index size and the query answering times for Hercules depend on the data distribution [135], we provide best and worst case asymptotic analysis.

Consider an index over a dataset of size N such that each index leaf contains at most th series ($th \ll N$). Note that Hercules fits the entire index in memory with leaves pointing to the raw data on-disk and that it can produce an unbalanced index tree.

Space Complexity

Best Case: The best case occurs when the Hercules binary tree index has the smallest possible number of nodes, i.e., the index tree is a full and complete binary tree. In this case, the index will have a total of $\lceil \frac{N}{th} \rceil$ leaves. Since the index is full, the total number of nodes in the index will be $2\lceil \frac{N}{th} \rceil - 1$.

Worst Case: The worst case is when the index has the largest possible number of nodes. This can happen when each leaf contains only one series, except for one leaf with th series, as a result of each new series insertion causing a leaf split where only one series ends up in one of the children. Therefore, the index tree will have $N - th + 1$ leaves. Since the index is full, the total number of nodes in the index will be $2(N-th)+1$. (Note that this is a pathological case that would happen when all series are almost identical: in this case, indexing and similarity search are not useful anyways.)

Time Complexity

As we consider large on-disk datasets, the query runtime is I/O bound; thus we express complexity in terms of I/O [211, 212], using the dataset size N, the index leaf threshold th and the disk block size B. We count one disk page access of size B as one I/O operation (for simplicity, we use B to denote the number of series that fit in one disk

page).

Best Case. The best case scenario occurs when one of the children of the index root is a leaf, containing one data series. In this case, the approximate search will incur $\Theta(1)$ I/O operation. In the best case, exact search will prune all other nodes of the index and thus will also incur $\Theta(1)$ disk access.

Worst Case. Approximate search always visits one leaf. Therefore, the worst case occurs when the leaf is the largest possible, i.e., it contains th series, in which case approximate search incurs $\Theta(th/B)$ I/O operations. For exact search, the worst case occurs when the algorithm needs to visit every single leaf of the index and the index has the maximum possible number of leaves. This can happen in the worst case we discussed above for index space complexity where the index tree has $N - th + 1$ leaves. Therefore, the exact search algorithm will access all the leaves, and will incur $\Theta(N)$ I/O operations. (Note again that this is a pathological case where indexing and similarity search are not useful anyways.)

5.3.4 Proofs

Consider definitions 3, 5 and 9 from Chapter 2, that we reproduce below for the sake of clarity.

Definition 3. *Consider an embedding f from ($\mathbb{M}$,d) to ($\mathbb{V}$,d'), where d and d' are the distance metrics in the original and embedding spaces respectively, the distance d' is a* ***lower-bounding*** *distance if:* $\forall\ S_Q, S_C \in \mathbb{M}\ \ d'(f(S_Q), f(S_C)) \leq d(S_Q, S_C)$ *[124].*

Definition 5. *Given an integer k, a* ***k-NN query*** *retrieves the set of objects* $\mathbb{A} = \{\{S_{C_1}, ..., S_{C_k}\} \subseteq \mathbb{S} | \forall\ S_C \in \mathbb{A}$ *and* $\forall\ S_{C'} \notin \mathbb{A},\ d(S_Q, S_C) \leq d(S_Q, S_{C'})\}$.

Definition 9. *Given a query S_Q, and $\epsilon \geq 0$, an* ***ϵ-approximate*** *algorithm guarantees that all results, S_C, are at a distance $d(S_Q, S_C) \leq (1+\epsilon) d(S_Q, [k\text{-th NN of } S_Q])$ in the case of a k-NN query, and distance $d(S_Q, S_C) \leq (1+\epsilon) r$ in the case of an r-range query.*

A Hercules index H over a dataset $\mathbb{M}$, consisting of a binary tree H_T, an $LRDFile$ H_R and an $LSDFile$ H_S, has the following properties:

Property 1. *Each node $N \in H_T$ is associated with its own embedding f_N from $(\mathbb{M}, d)$ to $(\mathbb{V}, d')$, where d' is the EAPCAMinDist [25].*

Property 2. *$\forall$ $S_C \in \mathbb{M}$, $\exists$ a path $\alpha \in H_T$ connecting the root node of H_T to one leaf node in H_T containing S_C. Given a query object S_Q, $\forall$ nodes $N_i \in \alpha$, $EAPCAMinDist(f_{N_i}(S_Q), f_{N_i}(S_C)) \leq d(S_Q, S_C)$.*

Property 3. *$\forall$ $S_C \in \mathbb{M}$, $\exists$ a vector $V_C \in H_S$ such that $V_C = SAX(S_C)$. Given a query object S_Q, $\forall$ $V_C \in H_S, SAXMinDist(PAA(S_Q), V_C) \leq d(S_Q, S_C)$ [146].*

Theorem 1. *Given a query S_Q and a Hercules index H, Algorithm 8 returns the set $\mathbb{A} = \{S_{C_1}, ..., S_{C_k}\}$ containing the kNN neighbors of S_Q.*

Proof. We will prove Theorem 1 by contradiction for $k = 1$. We skip the proof for an arbitrary k because it is straightforward to prove by induction.

Assume Algorithm 8 does not return S_C the NN of S_Q; therefore, S_C must have been pruned at either line 7, 9, 10 or 11.

Case 1: S_C was pruned at line 7 of Algorithm 8

This means that it was pruned by object S'_C which is not a NN of S_Q and which satisfies $d(S'_C, S_Q) = kthbsf$ at either line 5, 11 or 21 of Algorithm 9. At line 5, either S_C belongs to the subtree rooted at node n or at a node n' still in *pqueue*, and knowing that the priority in *pqueue* is based on the lowest value of the $EAPCAMinDist$, we have $EAPCAMinDist(S_Q, n') \geq EAPCAMinDist(S_Q, n) > kthbsf$. It follows from property 2 that $d(S_Q, S_C) > kthbsf$. Per Definition 5, this cannot happen since S_C, not S'_C, is the NN of S_Q. S_C could not have been pruned at line 11 because it would mean that $d(S_Q, S_C) \geq kthbsf$ and from Definition 5 that $d(S_Q, S_C) = kthbsf$ which contradicts our assumption that S'_C is not the NN of S_Q. Similarly, S_C cannot be pruned at line 21

because the subtree rooted at $childNode$ and containing S_C would satisfy the condition $d(S_Q, S_C) \geq EAPCAMinDist(S_Q, n) \geq kthbsf$. Therefore Case 1 is not plausible.

Case 2: S_C was pruned at line 9 of Algorithm 8

This means that it was pruned at line 4 or 12 of Algorithm 10. Following the same reasoning as in Case 1, this scenario is not plausible either.

Case 3: S_C was pruned at line 10 of Algorithm 8

This could have happened only if it S_C was not added to any $VCList$ because it was pruned at line 6 of Algorithm 12, i.e., we have $SAXMinDist(PAA(S_Q), S_C) \geq kthbsf$. It follows from Property 3 that $d(S_Q, S_C) \geq kthbsf$ and from Definition 5 that $d(S_Q, S_C) = kthbsf$ which contradicts our assumption that S'_C is not the NN of S_Q.

Case 4: S_C was pruned at line 11 of Algorithm 8

This means that it was pruned at line 6 of Algorithm 14, i.e., that $d(S_Q, S_C) \geq kthbsf$. It follows from Definition 5 that $d(S_Q, S_C) = kthbsf$ which contradicts our assumption that S'_C is not the NN of S_Q.

Since neither cases 1-4 can occur, this contradicts our original assumption.

□

Theorem 2. *Given a query S_Q, a Hercules index H, $\epsilon \geq 0$ and $\delta \in [0,1]$, Algorithm 15 returns the set $\tilde{\mathbb{A}} = \{\tilde{S}_{C_1}, ..., \tilde{S}_{C_k}\}$ containing the k δ-ϵ-approximate neighbors of S_Q.*

Proof. . We will prove Theorem 2 for $\delta = 1$. For $0 \leq \delta < 1$, a complete proof has appeared in [18, 198].

Case 1: $\delta = 1$

We will prove this case for $\delta = 1$ and $k = 1$ by contradiction. We skip the proof for an arbitrary k because it is straightforward to prove by induction.

Assume Algorithm 15 does not return $\tilde{S}_C$ the ϵ-approximate NN of S_Q; therefore, $\tilde{S}_C$ must have been pruned at either line 9 or 12 (as lines 7 and 10 are the same as the exact algorithm for which we have already proven that they do not cause any false dismissals). Also consider that S_C is the exact NN of S_Q.

If $\tilde{S}_C$ is pruned at line 9 of Algorithm 15, this means that it is pruned by object S'_C, which satisfies $d(S_Q, S'_C) = kthbsf$ and is not an ϵ-approximate NN of S_Q, at either line 4 or 12 of Algorithm 16. At line 4, either $\tilde{S}_C$ belongs to the subtree rooted at node n or at a node n' still in *pqueue*, and knowing that the priority in *pqueue* is based on the lowest value of the $EAPCAMinDist$, we have $EAPCAMinDist(S_Q, n') \geq EAPCAMinDist(S_Q, n) > kthbsf/(1+(\epsilon))$. It follows from property 2 that $d(S_Q, \tilde{S}_C) > kthbsf/(1+\epsilon)$. Since S'_C cannot be closer to S_Q than S_C then $d(S_Q, \tilde{S}_C) > kthbsf/(1+\epsilon) \geq d(S_Q, S'_C)$ but this violates Definition 9 since $\tilde{S}_C$ is the ϵ-approximate neighbor of S_Q. Following the same argument $\tilde{S}_C$ cannot be pruned by S'_C at line 12.

If $\tilde{S}_C$ is pruned at line 12 of Algorithm 15, it means that the stop condition at line 11 of Algorithm 18 is activated. When $\delta = 1$, $r_\delta(Q) = -\infty$, therefore the stop condition is never activated.

This contradicts our assumption. Therefore, when $\delta = 1$, Algorithm 15 returns the the ϵ-approximate NN of S_Q.

Case 2: $0 \leq \delta < 1$

We refer the reader to the detailed proof in [18, 198].

□

5.4 Experimental Evaluation

We evaluated Hercules against the state-of-the-art similarity search methods per the findings of our extensive experimental studies [95, 96].

5.4.1 Environment

We compiled all methods with GCC 6.2.0 under Ubuntu Linux 16.04.2 with their default compilation flags; optimization level was set to 2. Experiments were run on a server with two Intel Xeon E5-2650 v4 2.2GHz CPUs, (30MB cache, 12 cores, 24 hyper-

threads), 75GB[2] of RAM, and 10.8TB (6 x 1.8TB) 10K RPM SAS hard drives in RAID0 with a throughput of 1290 MB/sec.

5.4.2 Experimental Framework

Algorithms. iSAX2+ [93], DSTree [93], ParIS [32] and VA+file [93] representing exact data series methods with support for approximate queries; and Faiss IMI [35], SRS [200], and QALSH [60] representing strictly approximate methods for vectors. We extended DSTree, iSAX2+, ParIS and VA+file with Algorithm 2 from Chapter 4, approximating r_δ with density histograms on a 100K data series sample, following the C++ implementation of [18]. All methods are single core implementations, except for Herucles, IMI and ParIS that make use of multi-threading and SIMD vectorization. We allow each method to leverage its full functionalities. and we use various metrics, including implementation-independent ones, to guard against bias. Our baseline is the Euclidean distance version of the UCR Suite [5]. This is a set of techniques for performing very fast similarity computation scans. These optimizations include: a) avoiding the computation of square root on Euclidean distance, b) early abandoning of Euclidean distance calculations, and c) reordering early abandoning on normalized data[3]. Data series points are represented using single precision values and methods based on fixed summarizations use 16 dimensions.

Datasets. We use synthetic and real datasets. Synthetic datasets, called *Rand*, were generated as random-walks using a summing process with steps following a Gaussian distribution (0,1). Such data model financial time series [124] and have been widely used in the literature [124, 28, 135]. Our three real datasets cover domains as varied as deep learning, seismology, and neuroscience. *Deep1B* [167] comprises 1 billion vectors of size 96 representing deep network embeddings extracted from the last layers of a convolutional neural network. To the best of our knowledge, the Deep1B dataset is the

[2]We used GRUB to limit the amount of RAM, so that all methods are forced to use the disk. Note that GRUB prevents the operating system from using the rest of the RAM as a file cache, which is what we wanted for our experiments.

[3]Early abandoning of Z-normalization is not used since all datasets were normalized in advance.

largest publicly available real dataset of deep network embeddings. *Seismic100GB* [140], contains 100 million data series of size 256 representing earthquake recordings at seismic stations worldwide. *Sald100GB* [166] contains neuroscience MRI data and includes 200 million data series of size 128. In our experiments, we vary the size of the datasets from 25GB to 250GB. The name of each dataset is suffixed with its size. We do not use other real datasets that have appeared in the literature [202, 187], because they are very small, not exceeding 1GB in size.

Queries. All our query workloads consist of 100 query series run asynchronously, i.e., not in batch mode. Synthetic queries were generated using the same random-walk generator as the *Rand* dataset (with a different seed, reported in [94]). For each dataset, we use two different query workloads: 1) a controlled workload, named with the suffix -*Ctrl*, generated by adding progressively larger amounts of noise to data series extracted from the raw dataset, so as to produce queries having different levels of difficulty, following the ideas in [149]; 2) a random workload, named with the suffix -*Rand*, that is generated by randomly selecting 100 queries from the raw dataset and excluding them during indexing. Since the *Deep*1*B* archive contains a real workload, the random queries for this dataset are selected from this workload. To assess Hercule's query answering against the different algorithms, we use the controlled query workloads for exact query answering and the random workloads for approximate query answering. The random workloads are typically hard and should not be used to assess the performance of index-based exact similarity search, since a sequential scan would be the method of choice on most datasets [95]. Our experiments cover ng-approximate and δ-ϵ-approximate k-NN queries, where k $\in [1, 100]$. We also include results for exact queries to serve as a yardstick. All datasets and queries are z-normalized to allow efficient similarity search [148].

Scenarios. Our experimental evaluation proceeds in four main steps: (i) we tune methods to their optimal parameters (§5.5.1); (ii) we evaluate the indexing scalability of the methods (§4.5.2); (iii) we compare in-memory and out-of-core scalability and accuracy

of all methods (§4.5.3-§4.5.4); and (iv) we perform additional experiments on the best performing methods for disk-resident data (§4.5.4).

Measures. We assess methods using the following criteria:

(1) Scalability and search efficiency using: *wall clock time* (input, output, CPU and total time), *throughput* (# of queries answered per minute), and two implementation-independent measures: the *number of random disk accesses* (# of disk seeks) and the *percentage of data accessed*.

(2) Search accuracy is assessed using: *Avg_Recall*, *Mean Average Precision (MAP)*, and *Mean Relative Error (MRE)*. Recall is the most commonly used accuracy metric in the approximate similarity search literature. However, since it does not consider rank accuracy, we also use MAP [203] that is popular in information retrieval [204, 205] and has been proposed recently in the high-dimensional community [170] as an alternative accuracy measure to recall. For a workload of queries $S_{Q_i} : i \in [1, N_Q]$, these are defined as follows.

1. $Avg_Recall(workload) = \sum_{i=1}^{N_Q} Recall(S_{Q_i})/N_Q$
2. $MAP(workload) = \sum_{i=1}^{N_Q} AP(S_{Q_i})/N_Q$
3. $MRE(workload) = \sum_{i=1}^{N_Q} RE(S_{Q_i})/N_Q$

where:

- $Recall(S_{Q_i}) = \frac{\#\ true\ neighbors\ returned\ by\ Q_i}{k}$
- $AP(S_{Q_i}) = \frac{\sum_{r=1}^{k}(P(S_{Q_i},r) \times rel(r))}{k}, \forall i \in [1, N_Q]$
  - $P(S_{Q_i}, r) = \frac{\text{\# true neighbors among the first } r \text{ elements}}{r}$.
  - $rel(r)$ is equal 1 if the neighbor returned at position r is one of the k exact neighbors of S_{Q_i} and 0 otherwise.
- $RE(S_{Q_i}) = \frac{1}{k} \times \sum_{r=1}^{k} \frac{d(S_{Q_i}, S_{C_r}) - d(S_{Q_i}, S_{C_i})}{d(S_{Q_i}, S_{C_i})}$. S_{C_i} is the exact nearest neighbor of S_{Q_i} and

S_{C_r} is the r-th NN retrieved[4]. Without loss of generality, we do not consider the case

[4] Note that in Definition 9, ϵ is an upper bound on $RE(S_{Q_i})$.

where $d(S_{Q_i}, S_{C_i}) = 0$. (i.e., range queries with radius zero, or kNN queries where the 1-NN is the query itself[5].)

(3) Size, using the *main memory* footprint of the algorithm.

Procedure. Experiments involve two steps: index building and query answering. Caches are fully cleared before each step, and stay warm between consecutive queries. For large datasets that do not fit in memory, the effect of caching is minimized for all methods. All experiments use workloads of 100 queries. Results reported for workloads of 10K queries are extrapolated: we discard the 5 best and 5 worst queries of the original 100 (in terms of total execution time), and multiply the average of the 90 remaining queries by 10K.

5.5 Results

5.5.1 Parametrization

We start by fine tuning each method (graphs omitted for brevity). In order to understand the speed/accuracy tradeoffs, we fix the total memory size available to 75GB. The optimal parameters for DSTree, iSAX2+ and VA+file are set according to [95] and those for ParIS are chosen per [32]. For indexing, the buffer and leaf sizes are set to 60GB and 100K, respectively, for both DSTree and iSAX2+. iSAX2+ is set to use 16 segments. VA+file uses a 20GB buffer and 16 DFT symbols. ParIS uses a 20GB buffer and a 2K leaf size. For SRS, we set M (the projected space dimensionality) to 16 so that the representations of all datasets fit in memory. The settings were the same for all datasets. The fine tuning for IMI proved more tricky and involved many testing iterations since the index building parameters strongly affect the speed/accuracy of query answering and differ greatly across datasets. For this reason, different parameters were chosen for different datasets. To tune the Faiss implementation of IMI, we followed

[5] In these cases, the MRE definition can be extended to use the symmetric mean absolute percentage error [206].

the guidelines in [35]. For the in-memory datasets, we set the index factory key to PQ32_128,IMI2x12,PQ32 and the training size to 1,048,576 vectors, while for disk based datasets, the index key is PQ32_128,IMI2x14,PQ32 and the training size is 4,194,304 vectors. To tune δ-ϵ-approximate search performance and accuracy, we vary δ and ϵ for SRS and ϵ for DSTree, iSAX2+, ParIS and the VA+file. For ng-approximate search, we vary the *nprobe* parameter for DSTree/iSAX2+/IMI/ParIS/VA+file (*nprobe* represents the number of visited leaves for DSTree/iSAX2+, the number of visited raw series for ParIS and VA+file, and the number of inverted lists for IMI).

5.5.2 Exact Query Answering

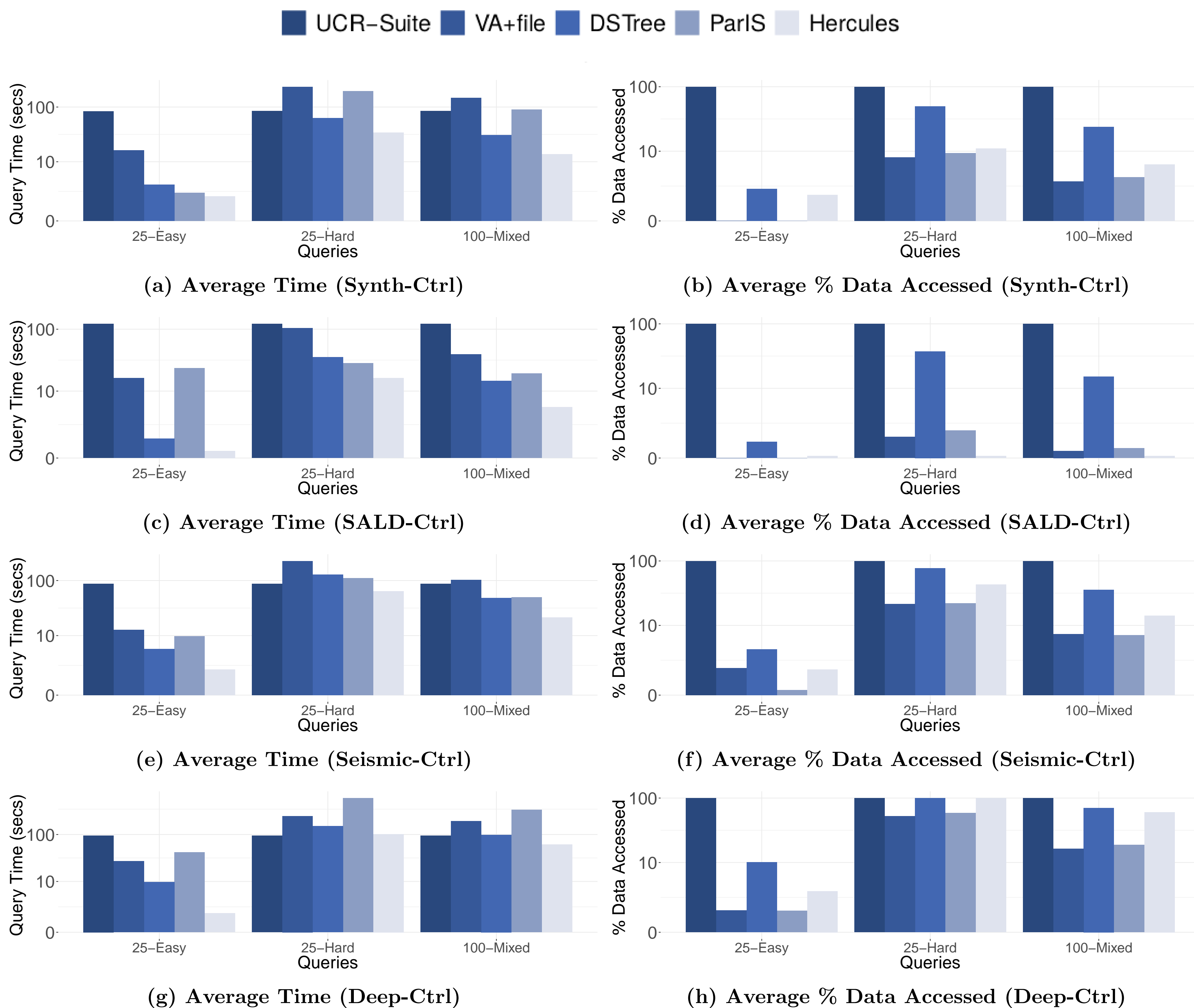
Figure 5.7 demonstrates the superiority of Hercules in exact query answering across the synthetic and real datasets. We use controlled workloads to show how each technique performs with queries having different levels of difficulty. We can observe that Hercules is the best performer overall.

On the complete workload containing 100 queries of increasing difficulty, Hercules is at least an order of magnitude faster than a sequential scan and 4 times faster than the state-of-the-art parallel index ParIS on the synthetic and Sald datasets (Figures 5.7a and 5.7c). As for the Seismic dataset, Hercules is at least 4 times faster than all competing methods (Fig. 5.7e). On the Deep dataset, which is notoriously hard [35, 159, 95, 96], Hercules still outperforms a sequential scan, which is not the case for any of the indexes (Fig. 5.7-g).

On the easy queries, Hercules outperforms the other methods by an even larger margin. It is at least one and a half order of magnitude faster than a sequential scan on all datasets and 4 times faster than the best index on the real datasets. Hercules leverages the double filtering scheme to prune more data than the DSTree as can be observed in Figures 5.7b, 5.7d, 5.7f, 5.7h.

On the hard queries, Hercules outperforms all the indexes. In the case of the hard

dataset Deep, for which an index is not the right access method anyways [95], all indexes degenerate in comparison to a sequential scan, except Hercules which performs equally well (Fig. 5.7g). This is despite the fact that Hercules accesses almost 100% of the data (Fig. 5.7h). The reason behind this is two-fold: 1) Hercules takes advantage of the LRD index storage layout which consists of only one file containing all the leaves data; and 2) all the leaf accesses are organized by their positions in the LRD file. Therefore Hercules' I/O accesses are treated by the disk as sequential.



(a) Average Time (Synth-Ctrl)

(b) Average % Data Accessed (Synth-Ctrl)

(c) Average Time (SALD-Ctrl)

(d) Average % Data Accessed (SALD-Ctrl)

(e) Average Time (Seismic-Ctrl)

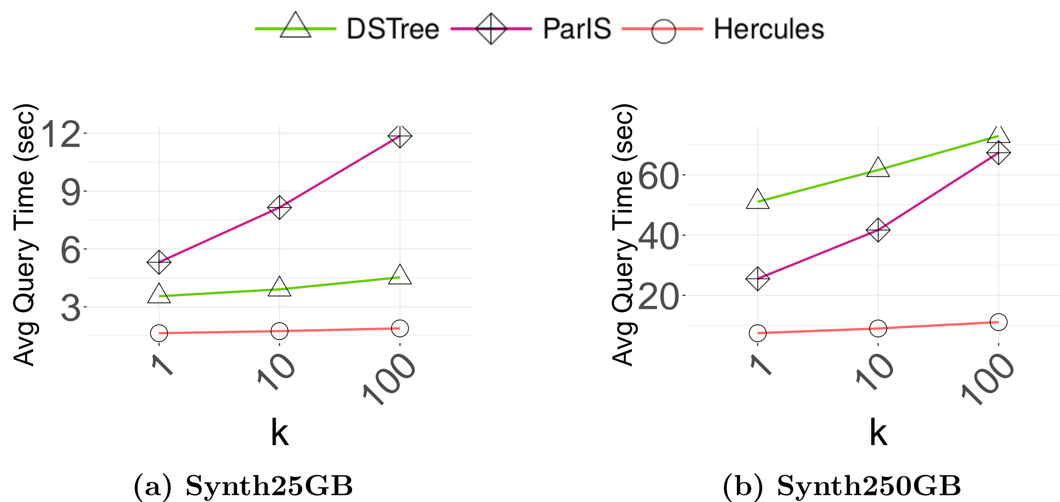
(f) Average % Data Accessed (Seismic-Ctrl)

(g) Average Time (Deep-Ctrl)

(h) Average % Data Accessed (Deep-Ctrl)

Figure 5.7: Comparison of exact query answering using 1NN queries of different difficulty

Based on the results in Figure 5.7, we selected the best methods (DSTree, ParIS and Hercules) and conducted further experiments varying the number of neighbors k in the query workload. Figure 5.8 summarizes the results of these experiments both in-memory (Rand25GB) and on-disk (Rand250GB). We measure the total time required to complete a workload of 100 queries for each value of k. We observe that Hercules wins across the board for all values of k. The figure also illustrates that finding the first neighbor is the most costly operation for DSTree and Hercules, while the performance of ParIS deteriorates as the number of neighbors increases. We believe this is due to ParIS being a skip sequential algorithm therefore the neighbors of a query can be located anywhere in the dataset file, whereas DSTree and Hercules are tree-based indexes so the neighbors tend be in the same subtree.



(a) Synth25GB **(b) Synth250GB**

Figure 5.8: Exact query answering for best methods with increasing k

5.5.3 Approximate Query Answering

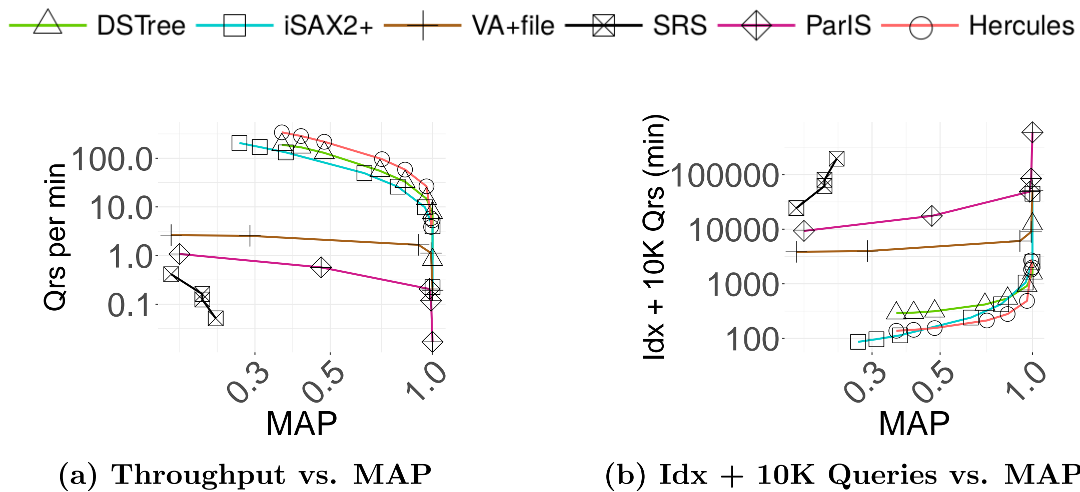
To evaluate the performance of Hercules in approximate search, we ran two sets of experiments, the first with δ-ϵ-approximate queries and the second with ng-approximate queries.

δ-ϵ-Approximate Query Answering

We first ran δ-ϵ-approximate queries on all the methods that support this type of search using a synthetic disk-based dataset of size 250GB. We can observe that Hercules is the best performer for all levels of accuracy in terms of throughput (Fig. 5.8ba) and the combined time needed to index and answer 10K queries (Fig. 5.9).

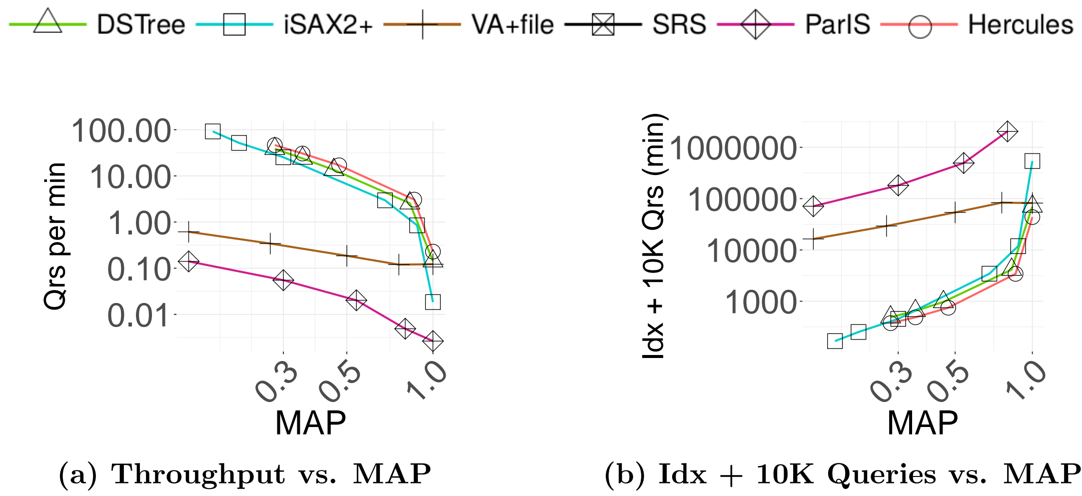
We can observe a similar behavior on the Deep250GB dataset (Fig. 5.10) where Hercules is again the winner for all values of MAP. Note that due to severe swapping issues, we could not run this experiment for the LSH-based method SRS.

We selected the best methods, i.e., DSTree, iSAX1+, ParIS and Hercules, and ran the same experiment with two additional datasets. The results on the Seismic100GB (Fig. 5.11) and Sald100GB (Fig. 5.12) datasets further confirm the superiority of Hercules in δ-ϵ-approximate query answering.



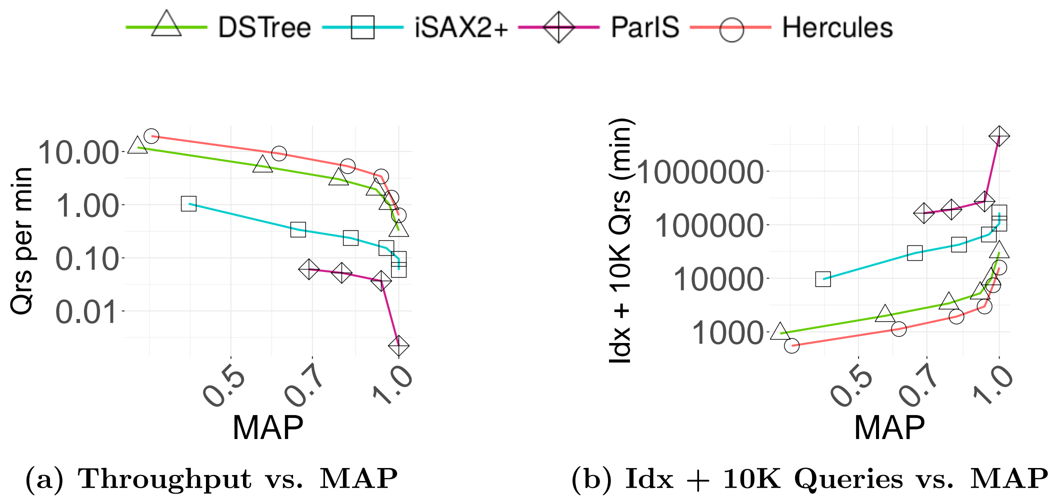
(a) Throughput vs. MAP **(b) Idx + 10K Queries vs. MAP**

Figure 5.9: δ-ϵ-approximate search (Dataset= Synth250GB, Queries = 100NN)



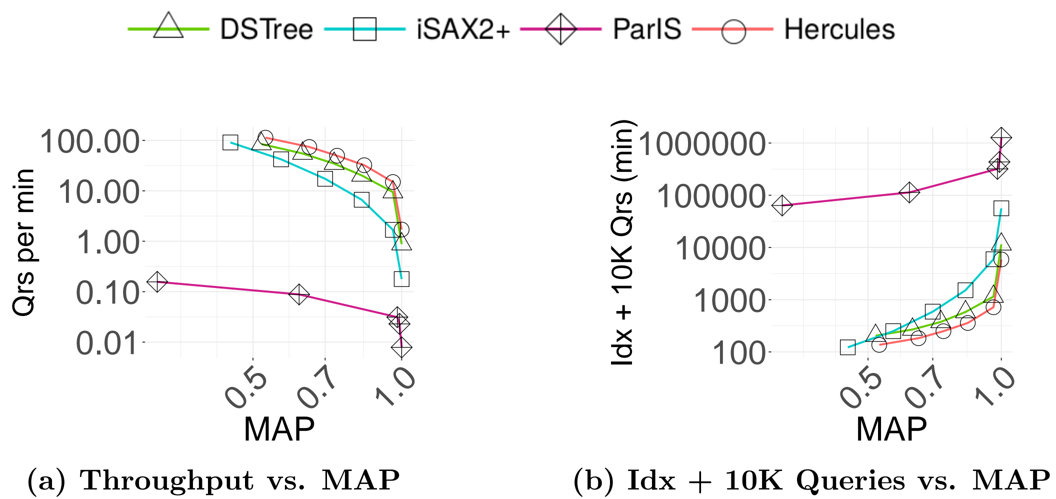
(a) **Throughput vs. MAP** (b) **Idx + 10K Queries vs. MAP**

Figure 5.10: δ-ϵ-approximate search
(Dataset= Deep250GB, Queries = 100NN)



(a) **Throughput vs. MAP** (b) **Idx + 10K Queries vs. MAP**

Figure 5.11: δ-ϵ-approximate search
(Dataset= Seismic100GB, Queries = 100NN)



(a) Throughput vs. MAP **(b) Idx + 10K Queries vs. MAP**

Figure 5.12: δ-ϵ-approximate search (Dataset= Sald100GB, Queries = 100NN)

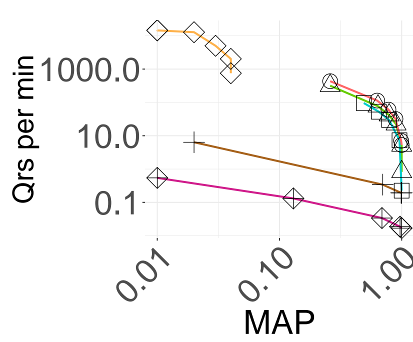
ng-Approximate Query Answering

We now evaluate the performance of Hercules on *ng*-approximate queries. Note that we only include disk-based methods since the datasets do not fit in memory.

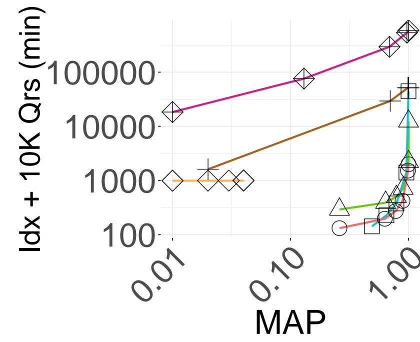
Figure 5.13 shows that Hercules outperforms all the other methods in terms of throughput and the combined time of indexing and answering 10K queries. IMI has a higher throughput for very low accuracy which is not desirable in practice. It is also noteworthy to point out that IMI does not incur any random I/O because it compresses all the raw data and stores the summaries in memory, whereas DSTree, iSAX2+ and Hercules use the memory to store their data structures, but read the raw data from disk.

We repeat the same experiment with the Deep250GB, Seismic100GB and Sald100GB datasets excluding the under performing methods IMI. VA+file and ParIS. We observe again that Hercules is the winner overall (Figures 5.14- 5.16).

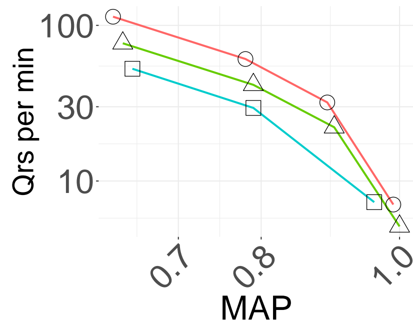
DSTree iSAX2+ VA+file IMI ParIS Hercules



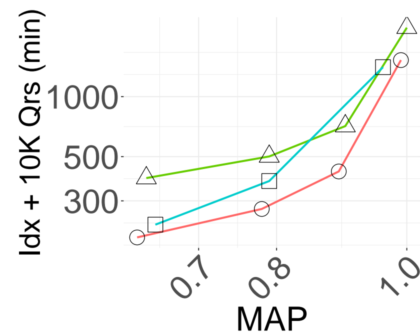
(a) **Throughput vs. MAP**



(b) **Idx + 10K Queries vs. MAP**



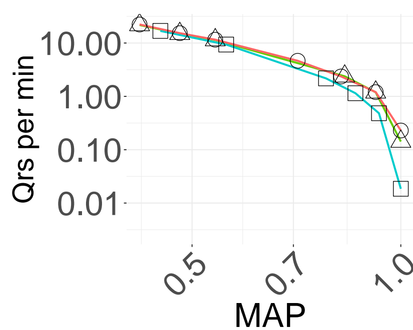
(c) **Throughput vs. MAP (best)**



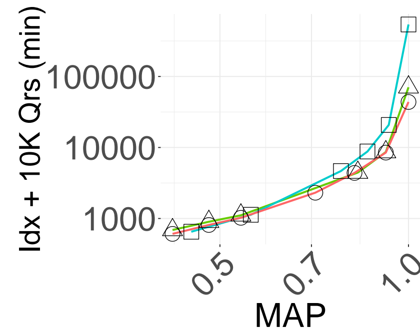
(d) **Idx + 10K Queries vs. MAP (best)**

Figure 5.13: ***ng*-approximate search**
(Dataset= Synth250GB, Queries = 100NN)

DSTree iSAX2+ Hercules

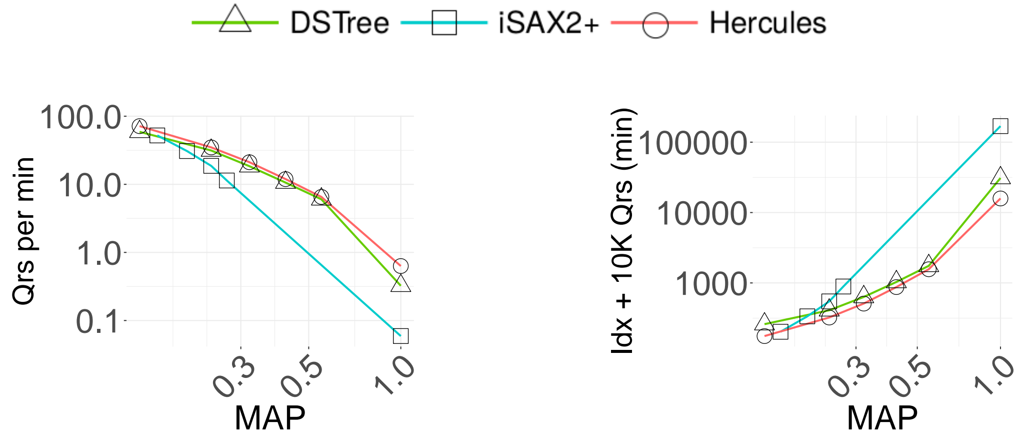


(a) **Throughput vs. MAP**



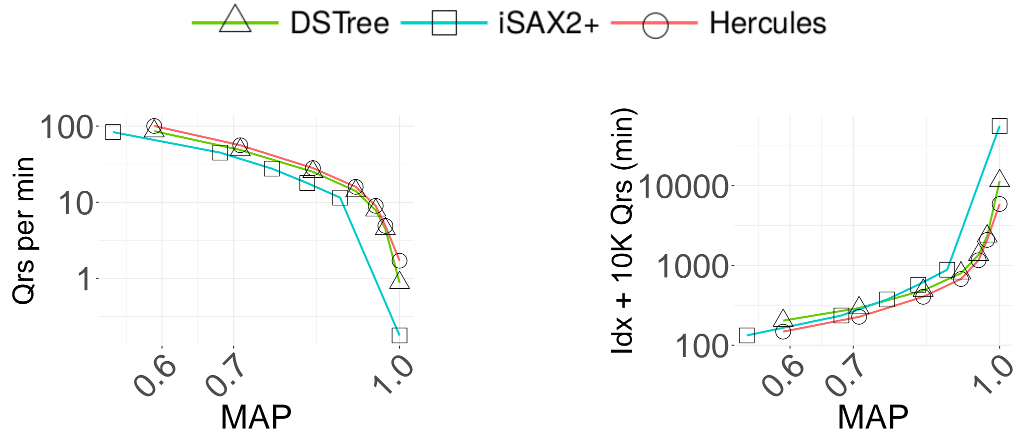
(b) **Idx + 10K Queries vs. MAP**

Figure 5.14: ***ng*-approximate search**
(Dataset= Deep250GB, Queries = 100NN)



(a) Throughput vs. MAP (best) **(b) Idx + 10K Queries vs. MAP (best)**

Figure 5.15: *ng*-approximate search (Dataset= Seismic100GB, Queries = 100NN)



(a) Throughput vs. MAP (best) **(b) Idx + 10K Queries vs. MAP (best)**

Figure 5.16: *ng*-approximate search (Dataset= Sald100GB, Queries = 100NN)

5.6 Conclusions

In this chapter we introduced Hercules, a new similarity search technique that leverages the deep insights gained from our two experimental evaluations about the intricate designs of the different techniques, their strengths and weaknesses. Our thorough experiments demonstrate that Hercules outperforms our proposed extensions in [96] and the state-of-the-art exact and approximate similarity search techniques in-memory and on-disk. We used synthetic datasets and 4 of the largest publicly available real datasets

of high-dimensional vectors, including data series and deep neural network embeddings.

The key features behind the superior performance of Hercules are the following novel ideas: 1) a double-filtering scheme which significantly reduces the number of surviving candidates compared to competing techniques; 2) an efficient storage mechanism for the index; 3) the carefully crafted distribution and execution of parallel instructions; and 4) the exploitation of the Single Instruction Multiple Data (SIMD) capabilities of modern CPUs.

Chapter 6

Conclusions and Future Work

Similarity search is a fundamental operation that lies at the core of many critical data processing tasks, such as data cleaning, data integration, and big data analytics (e.g., outlier detection, frequent pattern mining, clustering, and classification). Similarity search aims at finding objects in a collection that are close to a given query according to some definition of sameness.

This problem has been studied heavily in the past 25 years and will continue to attract attention as massive collections of high-dimensional objects are becoming omnipresent in various domains. A number of exact and approximate approaches have been proposed in the literature to support similarity search over massive data series collections.

This chapter summarizes the main contributions of this thesis in solving the similarity search problem (§ 6.1) and concludes by pinpointing promising future research directions that go beyond addressing limitations of our work (§ 6.2).

6.1 Key Contributions

In this work, we unified and formally defined the terminology used for the different flavors of the similarity search problem. We presented a similarity search taxonomy that classifies methods based on the quality guarantees they provide for the search results, and that unifies the varied nomenclature used in the literature. Following this taxonomy, we included a survey of similarity search approaches supporting exact and approximate search, bringing together works from the data series and high-dimensional data research communities.

We also designed and executed two thorough experimental comparisons of several relevant techniques in the literature which had never before been compared at equal footing to one another. Our experimental evaluations are the most comprehensive in the literature and paint a clear picture of the strengths and weaknesses of the various approaches, uncovering very interesting insights and indicating promising research directions. The findings of both studies lay the ground for solid further developments in the field of similarity search.

We proposed extensions to existing data series indexes that can answer δ-ϵ-approximate queries and that outperform popular state-of-the-art techniques such as LSH, kNN graphs and quantization-based inverted indexes. This is an important contribution that opens up a new research direction in the field of approximate similarity search, which has been extensively studied in the past 25 years by different communities following two main trends: (i) LSH-based algorithms that support guarantees, but are relatively slow, and (ii) kNN graphs and inverted indexes, which are relatively fast, but do not provide theoretical guarantees. Our work demonstrates that it is possible to have the best of both worlds and design efficient high-dimensional vector similarity search algorithms with theoretical guarantees on the quality of the answers. We thus offer a more promising alternative to the two current trends in the literature. This finding sets the stage for developing solutions that truly support scalable analytics.

A forward step in this direction is Hercules, a new similarity search technique that outperforms the state-of-the-art approximate and exact similarity search methods, including our own extensions to data series indexes. Hercules builds upon the insights gained from our two experimental studies about the intricate designs of the existing approaches, their strengths and weaknesses. Hercules performs equally well on massive collections of data series and high-dimensional vectors such as deep network embeddings. The interest for such work will continue to grow as deep network embeddings are becoming a big consumer of similarity search algorithms [84].

6.2 Future Work

An important contribution of this thesis is pinpointing promising research directions in the field of similarity search. Below, I outline some of these directions that go beyond addressing shortcomings of our proposed work.

Facilitating real data science. I would like to integrate Hercules in a real data management system to support data cleaning, data integration, and big data analytics (e.g., outlier detection, frequent pattern mining, clustering, and classification). This will not only enhance the impact of my current research results, but also spur new research avenues for extending the capabilities of Hercules.

Developing a benchmark for similarity search. Currently, there exists no benchnmark for similarity search. My goal is to extend the Lernaean Hydra Archives [93, 94] to serve as a benchmark that will help the research community standardize the evaluation of the quality and efficiency of similarity search techniques.

Improving ng-approximate search in-memory. While Hercules is the overall winner on-disk, for exact and approximate search, and in-memory for exact and δ-ϵ-approximate search, it is outperformed in-memory by the kNNG method HNSW. I plan to improve the performance of Hercules for memory-resident data, so that it becomes the method of

choice across all flavors of the similarity search problem.

Improving user interactivity with progressive query answering: Although Hercules is outperforming the current state-of-the-art techniques in exact search. Its query answering time is still not satisfactory for interactive analytics. I would like to enhance Herculess query answering algorithm to return progressive estimates of the final answer with probability guarantees. This will better support interactive exploration and fast decision making [209, 100].

Building a height-balanced index tree. Although empirically, the index building algorithm of Hercules produces a tree that is well balanced, i.e., most leaves have the same depth, this is not theoretically guaranteed. My objective is to modify the index building algorithm so that it guarantees a height-balanced tree, i.e., a tree where no leaf is farther away from the root than any other leaf. This will be useful to improve the worst-case complexity of the algorithm and establish average-case guarantees which can be leveraged to develop a cost model for Hercules.

Developing more effective stopping criteria. LSH techniques exploit both δ and ϵ to tune the efficiency/accuracy tradeoff. We consider that they are still inadequate because they can produce inaccurate results when ϵ is low and δ is high and lack in efficiency when δ is low. We have shown that, in the case of our Hercules and our extended methods, using ϵ yields excellent empirical results, but introducing the probabilistic stop condition based on δ was ineffective. We believe that this is due to the inaccuracy of the (histogram-based) approximation of r_δ. Therefore, improving the approximation of r_δ, or devising novel approaches are interesting open research directions that will further improve the efficiency of these methods.

Learning the segmentation of high-dimensional vectors. Hercules currently intertwines indexing and data segmentation using statistical information about the data and some heuristics to establish the quality of a given segmentation. I would like to explore whether machine learning techniques can help Hercules improve its segmentation

for datasets from various domains (data series, images, deep network embeddings) and how to establish lower-bounds and build an index tree using the learned segmentation.

Extending guarantees. In the approximate search literature, accuracy has been evaluated using recall, and distance approximation error. The existing techniques that provide guarantees base their guarantees on the approximation error. However, we established that recall and MAP are better indicators of accuracy, because even small approximation errors may still result in low recall/MAP values. Therefore, a promising research direction would be to develop probabilistic or deterministic guarantees on the recall or MAP values of a result set, instead of the commonly used distance approximation error.

Developing a cost model. Currently, there exists no cost model for any of the data series indexes, including Hercules. A cost model predicts the performance of similarity search query by estimating the I/O and CPU costs of an incoming query. Devising such a model is critical not only to establish a theoretical complexity analysis of the average query execution time, but also for query optimization and index parameter tuning.

Supporting uncertain data. My work has focused on data that is considered correct and free from noise. I would like to explore how to efficiently support similarity search on uncertain high-dimensional vector data, i.e., vectors that have missing values in some dimensions or contain errors. For instance, data uncertainty can manifest when IoT sensors produce imprecise measurements or as a result of privacy-preserving transformations.

Answering variable-length queries. The scope of my previous work was with fixed-length queries, that is the query vector and the vectors indexed in the dataset have the same number of dimensions. It would be very interesting to study how to support efficient exact and approximate similarity search in the presence of vectors of a variable number of dimensions. This is a common scenario in practice, for example, a neuroscientist might be interested in analyzing Electroencephalogram recordings of different lengths to study brain activity.

Automatic parameter tuning. To the best of our knowledge, all the state-of-the-

art similarity search methods require manual tuning. This is particularly problematic for approximate similarity search techniques based on LSH, k-NN graphs and inverted indexes because tuning these methods is cumbersome and time-consuming [96]. For instance, QALSH [60] is considered the state-of-the-art LSH method in terms of accuracy. However, it achieves this at a very high cost: it needs to build a different index for each desired query accuracy. This is a serious drawback that concerns also IMI and HSNW, which are considered among the best ng-approximate methods. The fact that the speed-accuracy tradeoff is not determined only at query time, but also during index building, means that an index may need to be built many times, using different parameters, before finding the right speed-accuracy tradeoff. Besides, the optimal settings may differ across datasets, and even across different dataset sizes for the same dataset. Developing auto-tuning mechanisms for these techniques is both an interesting problem and a necessity.

Bibliography

[1] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. The data civilizer system. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[2] Renée J. Miller. Open data integration. *PVLDB*, 11(12):2130–2139, 2018.

[3] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41(3):15, 2009.

[4] Michele Dallachiesa, Themis Palpanas, and Ihab F. Ilyas. Top-k Nearest Neighbor Search in Uncertain Data Series. *PVLDB*, 8(1):13–24, 2014.

[5] Thanawin Rakthanmanon, Bilson J. L. Campana, Abdullah Mueen, Gustavo E. A. P. A. Batista, M. Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn J. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In Qiang Yang, Deepak Agarwal, and Jian Pei, editors, *KDD*, pages 262–270. ACM, 2012.

[6] Thanawin Rakthanmanon, Eamonn J Keogh, Stefano Lonardi, and Scott Evans. Time series epenthesis: Clustering time series streams requires ignoring some data. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 547–556. IEEE, 2011.

[7] T. Warren Liao. Clustering of time series dataa survey. *Pattern Recognition*, 38(11):1857–1874, 2005.

[8] Yihua Chen, Eric K. Garcia, Maya R. Gupta, Ali Rahimi, and Luca Cazzanti. Similarity-based classification: Concepts and algorithms. *J. Mach. Learn. Res.*, 10:747–776, June 2009.

[9] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is nearest neighbor meaningful? In *Proceedings of the 7th International Conference on Database Theory*, ICDT 99, page 217235, Berlin, Heidelberg, 1999. Springer-Verlag.

[10] Vladimir Pestov. On the geometry of similarity search: Dimensionality curse and concentration of measure. *Information Processing Letters*, 73(1):47 – 51, 2000.

[11] Charu C. Aggarwal, Alexander Hinneburg, and Daniel A. Keim. On the surprising behavior of distance metrics in high dimensional spaces. In *Proceedings of the 8th International Conference on Database Theory*, ICDT 01, page 420434, Berlin, Heidelberg, 2001. Springer-Verlag.

[12] Damien Francois, Vincent Wertz, and Michel Verleysen. The concentration of fractional distances. *IEEE Trans. on Knowl. and Data Eng.*, 19(7):873886, July 2007.

[13] Richard E. Bellman. *Adaptive Control Processes: A Guided Tour*. MIT Press, 1961.

[14] Kristin P. Bennett, Usama Fayyad, and Dan Geiger. Density-based indexing for approximate nearest-neighbor queries. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD 99, page 233243, New York, NY, USA, 1999. Association for Computing Machinery.

[15] Robert J. Durrant and Ata Kabn. When is nearest neighbour meaningful: A converse theorem and implications. *Journal of Complexity*, 25(4):385 – 397, 2009.

[16] Ata Kabán. Non-parametric detection of meaningless distances in high dimensional data. *Statistics and Computing*, 22(2):375–385, 2012.

[17] Junfeng He, Sanjiv Kumar, and Shih-Fu Chang. On the difficulty of nearest neighbor search. In *Proceedings of the 29th International Conference on International Conference on Machine Learning*, ICML12, page 4148, Madison, WI, USA, 2012. Omnipress.

[18] Paolo Ciaccia and Marco Patella. PAC Nearest Neighbor Queries: Approximate and Controlled Search in High-Dimensional and Metric Spaces. In *ICDE*, pages 244–255, 2000.

[19] and X. Sean Wang. Supporting content-based searches on time series via approximation. In *Proceedings. 12th International Conference on Scientific and Statistica Database Management*, pages 69–81, July 2000.

[20] Hakan Ferhatosmanoglu, Ertem Tuncel, Divyakant Agrawal, and Amr El Abbadi. Vector Approximation Based Indexing for Non-uniform High Dimensional Data Sets. In *Proceedings of the Ninth International Conference on Information and Knowledge Management*, CIKM '00, pages 202–209, New York, NY, USA, 2000. ACM.

[21] Richard Cole, Dennis E. Shasha, and Xiaojian Zhao. Fast window correlations over uncooperative time series. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Chicago, Illinois, USA, August 21-24, 2005*, pages 743–749, 2005.

[22] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. Efficient and Accurate Nearest Neighbor and Closest Pair Search in High-dimensional Space. *ACM Trans. Database Syst.*, 35(3):20:1–20:46, July 2010.

[23] Alessandro Camerra, Themis Palpanas, Jin Shieh, and Eamonn J. Keogh. iSAX 2.0: Indexing and Mining One Billion Time Series. In Geoffrey I. Webb, Bing Liu, Chengqi Zhang, Dimitrios Gunopulos, and Xindong Wu, editors, *ICDM*, pages 58–67. IEEE Computer Society, 2010.

[24] Shrikant Kashyap and Panagiotis Karras. Scalable kNN search on vertically stored time series. In Chid Apt, Joydeep Ghosh, and Padhraic Smyth, editors, *KDD*, pages 1334–1342. ACM, 2011.

[25] Yang Wang, Peng Wang, Jian Pei, Wei Wang, and Sheng Huang. A Data-adaptive and Dynamic Segmentation Index for Whole Matching on Time Series. *PVLDB*, 6(10):793–804, 2013.

[26] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. SRS: Solving c-approximate Nearest Neighbor Queries in High Dimensional Euclidean Space with a Tiny Index. *PVLDB*, 8(1):1–12, 2014.

[27] Abdullah Mueen, Yan Zhu, Michael Yeh, Kaveh Kamgar, Krishnamurthy Viswanathan, Chetan Gupta, and Eamonn Keogh. The Fastest Similarity Search Algorithm for Time Series Subsequences under Euclidean Distance, August 2017. `http://www.cs.unm.edu/~mueen/FastestSimilaritySearch.html`.

[28] Alessandro Camerra, Jin Shieh, Themis Palpanas, Thanawin Rakthanmanon, and Eamonn J. Keogh. Beyond one billion time series: indexing and mining very large time series collections with iSAX2+. *Knowl. Inf. Syst.*, 39(1):123–151, 2014.

[29] Yury A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *CoRR*, abs/1603.09320, 2016.

[30] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. ADS: the adaptive data series index. *The VLDB Journal*, 25(6):843–866, 2016.

[31] Michele Linardi and Themis Palpanas. ULISSE: ULtra compact Index for Variable-Length Similarity SEarch in Data Series. In *ICDE*, 2018.

[32] Botao Peng, Panagiota Fatourou, and Themis Palpanas. ParIS: The Next Destination for Fast Data Series Indexing and Query Answering. *IEEE BigData*, 2018.

[33] Djamel-Edine Yagoubi, Reza Akbarinia, Florent Masseglia, and Themis Palpanas. Massively distributed time series indexing and querying. *TKDE (to appear)*, 2019.

[34] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. Coconut: Sortable summarizations for scalable indexes over static and streaming data series. *VLDBJ*, accepted for publication, 2019.

[35] Faiss. `https://github.com/facebookresearch/faiss/`, 2019.

[36] Piotr Indyk and Tal Wagner. Approximate nearest neighbors in limited space. In *COLT*, 2018.

[37] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast Approximate Nearest Neighbor Search with the Navigating Spreading-out Graph. *PVLDB.*, 12(5):461–474, 2019.

[38] Botao Peng, Panagiota Fatourou, and Themis Palpanas. MESSI: In-Memory Data Series Indexing. *ICDE*, 2020.

[39] Sebastián Ferrada, Benjamin Bustos, and Nora Reyes. An efficient algorithm for approximated self-similarity joins in metric spaces. *Information Systems*, page 101510, 2020.

[40] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57, 1984.

[41] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[42] Patrick Schäfer and Mikael Högqvist. SFA: A Symbolic Fourier Approximation and Index for Similarity Search in High Dimensional Datasets. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 516–527, New York, NY, USA, 2012. ACM.

[43] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA*, pages 322–331. ACM, 1990.

[44] D. E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas. DPiSAX: Massively Distributed Partitioned iSAX. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 1135–1140, 2017.

[45] A. Babenko and V. Lempitsky. The Inverted Multi-Index. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(6):1247–1260, June 2015.

[46] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In Matthias Jarke, Michael Carey, Klaus R. Dittrich, Fred Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 426–435, Athens, Greece, August 1997. Morgan Kaufmann Publishers, Inc.

[47] H. Jegou, M. Douze, and C. Schmid. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, Jan 2011.

[48] H. Jegou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 861–864, May 2011.

[49] Yan Xia, Kaiming He, Fang Wen, and Jian Sun. Joint Inverted Indexing. *2013 IEEE International Conference on Computer Vision*, pages 3416–3423, 2013.

[50] Piotr Indyk and Rajeev Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[51] A. Broder. On the Resemblance and Containment of Documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES '97, pages 21–, Washington, DC, USA, 1997. IEEE Computer Society.

[52] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive Hashing Scheme Based on P-stable Distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, pages 253–262, New York, NY, USA, 2004. ACM.

[53] Moses S. Charikar. Similarity Estimation Techniques from Rounding Algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 380–388, New York, NY, USA, 2002. ACM.

[54] Ting Liu, Andrew W. Moore, Alexander Gray, and Ke Yang. An Investigation of Practical Approximate Nearest Neighbor Algorithms. In *Proceedings of the 17th International Conference on Neural Information Processing Systems*, NIPS'04, pages 825–832, Cambridge, MA, USA, 2004. MIT Press.

[55] Rina Panigrahy. Entropy Based Nearest Neighbor Search in High Dimensions. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, pages 1186–1195, Philadelphia, PA, USA, 2006. Society for Industrial and Applied Mathematics.

[56] Rajeev Motwani, Assaf Naor, and Rina Panigrahy. Lower Bounds on Locality Sensitive Hashing. *SIAM J. Discrete Math.*, 21(4):930–935, 2007.

[57] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe LSH: Efficient Indexing for High-dimensional Similarity Search. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 950–961. VLDB Endowment, 2007.

[58] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. Locality-sensitive Hashing Scheme Based on Dynamic Collision Counting. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 541–552, New York, NY, USA, 2012. ACM.

[59] Ryan O'Donnell, Yi Wu, and Yuan Zhou. Optimal Lower Bounds for Locality-Sensitive Hashing (Except When Q is Tiny). *ACM Trans. Comput. Theory*, 6(1):5:1–5:13, March 2014.

[60] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. Query-aware Locality-sensitive Hashing for Approximate Nearest Neighbor Search. *PVLDB*, 9(1):1–12, 2015.

[61] Sunil Arya and David M. Mount. Approximate Nearest Neighbor Queries in Fixed Dimensions. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 271–280, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[62] Edgar Chávez and Eric Sadit Tellez. Navigating K-Nearest Neighbor Graphs to Solve Nearest Neighbor Searches. In José Francisco Martínez-Trinidad, Jesús Ariel Carrasco-Ochoa, and Josef Kittler, editors, *Advances in Pattern Recognition*, pages 270–280, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[63] Kazuo Aoyama, Kazumi Saito, Hiroshi Sawada, and Naonori Ueda. Fast Approximate Similarity Search Based on Degree-reduced Neighborhood Graphs. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 1055–1063, New York, NY, USA, 2011. ACM.

[64] J. Wang, J. Wang, G. Zeng, R. Gan, S. Li, and B. Guo. Fast Neighborhood Graph Search Using Cartesian Concatenation. In *2013 IEEE International Conference on Computer Vision*, pages 2128–2135, Dec 2013.

[65] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61 – 68, 2014.

[66] Guillermo Ruiz, Edgar Chávez, Mario Graff, and Eric S. Téllez. Finding Near Neighbors Through Local Search. In *Proceedings of the 8th International Conference on Similarity Search and Applications - Volume 9371*, SISAP 2015, pages 103–109, Berlin, Heidelberg, 2015. Springer-Verlag.

[67] Zhansheng Jiang, Lingxi Xie, Xiaotie Deng, Weiwei Xu, and Jingdong Wang. Fast Nearest Neighbor Search in the Hamming Space. In *Proceedings, Part I, of the 22Nd International Conference on MultiMedia Modeling - Volume 9516*, MMM 2016, pages 325–336, Berlin, Heidelberg, 2016. Springer-Verlag.

[68] Themis Palpanas. Big Sequence Management: A glimpse of the Past, the Present, and the Future. In Rusins Martins Freivalds, Gregor Engels, and Barbara Catania, editors, *SOFSEM*, volume 9587 of *Lecture Notes in Computer Science*, pages 63–80. Springer, 2016.

[69] Kunio Kashino, Gavin Smith, and Hiroshi Murase. Time-series active search for quick retrieval of audio and video. In *ICASSP*, 1999.

[70] Dennis Shasha. Tuning Time Series Queries in Finance: Case Studies and Recommendations. *IEEE Data Eng. Bull.*, 22(2):40–46, 1999.

[71] Pavlos Paraskevopoulos, Thanh-Cong Dinh, Zolzaya Dashdorj, Themis Palpanas, and Luciano Serafini. Identification and Characterization of Human Behavior Patterns from Mobile Phone Data. In *D4D Challenge session, NetMob*, 2013.

[72] S Soldi, Volker Beckmann, WH Baumgartner, Gabriele Ponti, Chris R Shrader, P Lubiński, HA Krimm, F Mattana, and Jack Tueller. Long-term variability of AGN at hard X-rays. *Astronomy & Astrophysics*, 563:A57, 2014.

[73] Katsiaryna Mirylenka, Vassilis Christophides, Themis Palpanas, Ioannis Pefkianakis, and Martin May. Characterizing Home Device Usage From Wireless Traffic Time Series. In *EDBT*, pages 551–562, 2016.

[74] Pablo Huijse, Pablo A. Estévez, Pavlos Protopapas, Jose C. Principe, and Pablo Zegers. Computational Intelligence Challenges and Applications on Large-Scale Astronomical Time Series Databases. *IEEE Comp. Int. Mag.*, 9(3):27–39, 2014.

[75] Usman Raza, Alessandro Camerra, Amy L. Murphy, Themis Palpanas, and Gian Pietro Picco. Practical Data Prediction for Real-World Wireless Sensor Networks. *IEEE Trans. Knowl. Data Eng.*, 27(8), 2015.

[76] Martin Bach-Andersen, Bo Romer-Odgaard, and Ole Winther. Flexible Non-Linear Predictive Models for Large-Scale Wind Turbine Diagnostics. *Wind Energy*, 20(5):753–764, 2017.

[77] S. Knieling, J. Niediek, E. Kutter, J. Bostroem, C.E. Elger, and F. Mormann. An online adaptive screening procedure for selective neuronal responses. *Journal of Neuroscience Methods*, 291(Supplement C):36 – 42, 2017.

[78] Michele Linardi, Yan Zhu, Themis Palpanas, and Eamonn J. Keogh. Matrix Profile X: VALMOD - Scalable Discovery of Variable-Length Motifs in Data Series. 2018.

[79] Billy M. Williams and Lester A. Hoel. Modeling and forecasting vehicular traffic flow as a seasonal arima process: Theoretical basis and empirical results. *Journal of Transportation Engineering*, 129(6):664–672, 2003.

[80] Georges Hébrail. *Practical data mining in a large utility company*, pages 87–95. Physica-Verlag HD, Heidelberg, 2000.

[81] Anthony J. Bagnall, Richard L. Cole, Themis Palpanas, and Konstantinos Zoumpatianos. Data series management (dagstuhl seminar 19282). *Dagstuhl Reports*, 9(7):24–39, 2019.

[82] Themis Palpanas and Volker Beckmann. Report on the First and Second Interdisciplinary Time Series Analysis Workshop (ITISA). *ACM SIGMOD Record*, 48(3), 2019.

[83] The International Data Corporation (IDC). IDC Worldwide Global DataSphere IoT Device and Data Forecast. In *Doc #US45066919*, May 2019.

[84] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *arXiv preprint arXiv:1702.08734*, 2017.

[85] Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*. ACL, 2014.

[86] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[87] Daniel Smilkov, Nikhil Thorat, Charles Nicholson, Emily Reif, Fernanda B. Vigas, and Martin Wattenberg. Embedding projector: Interactive visualization and interpretation of embeddings, 2016.

[88] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 3111–3119, USA, 2013. Curran Associates Inc.

[89] Aditya Grover and Jure Leskovec. Node2vec: Scalable feature learning for networks. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 855–864, New York, NY, USA, 2016. ACM.

[90] A. B. Yandex and V. Lempitsky. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2055–2063, June 2016.

[91] Yingwei Pan, Yehao Li, Ting Yao, Tao Mei, Houqiang Li, and Yong Rui. Learning deep intrinsic video representation by exploring temporal coherence and graph structure. In *IJCAI*, pages 3832–3838, 2016.

[92] J. Wang, T. Zhang, j. song, N. Sebe, and H. T. Shen. A survey on learning to hash. *TPAMI*, 40(4), 2018.

[93] Lernaean Hydra Archive. `http://www.mi.parisdescartes.fr/~themisp/dsseval/`, 2018.

[94] Lernaean Hydra Archive II. `http://www.mi.parisdescartes.fr/~themisp/dsseval2/`, 2019.

[95] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art. *PVLDB*, 12(2):112–127, 2018.

[96] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. Return of the Lernaean Hydra: Experimental Evaluation of Data Series Approximate Similarity Search. *PVLDB*, 13(3):402–419, 2019.

[97] Karima Echihabi. Truly Scalable Data Series Similarity Search. In *VLDB PhD Workshop*, 2019.

[98] Karima Echihabi. High-Dimensional Similarity Search: From Time Series to Deep Network Embeddings. In *SIGMOD*, 2020.

[99] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, Panagiota Fatourou, and Houda Benbrahim. Hercules: Overcoming the Lernaean Hydra of High-Dimensional Similarity Search. Under Submission.

[100] Anna Gogolou, Theophanis Tsandilas, Karima Echihabi, Themis Palpanas, and Anastasia Bezerianos. Data Series Progressive Similarity Search with Probabilistic Quality Guarantees. In *SIGMOD*, 2020.

[101] J. Hafner, H. S. Sawhney, W. Equitz, M. Flickner, and W. Niblack. Efficient color histogram indexing for quadratic form distance functions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(7):729–736, July 1995.

[102] C. Faloutsos, W. Equitz, M. Flickner, W. Niblack, D. Petkovic, and R. Barber. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3:231–262, 1994.

[103] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004.

[104] Johannes Aßfalg, Hans-Peter Kriegel, Peer Kröger, and Matthias Renz. Probabilistic Similarity Search for Uncertain Time Series. In *Scientific and Statistical Database Management, 21st International Conference, SSDBM 2009, New Orleans, LA, USA, June 2-4, 2009, Proceedings*, pages 435–443, 2009.

[105] Mi-Yen Yeh, Kun-Lung Wu, Philip S. Yu, and Ming-Syan Chen. PROUD: a probabilistic approach to processing similarity queries over uncertain data streams. In *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, pages 684–695, 2009.

[106] Smruti R. Sarangi and Karin Murthy. DUST: a generalized notion of similarity between uncertain time series. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, July 25-28, 2010*, pages 383–392, 2010.

[107] Michele Dallachiesa, Besmira Nushi, Katsiaryna Mirylenka, and Themis Palpanas. Uncertain Time-Series Similarity: Return to the Basics. *PVLDB*, 5(11):1662–1673, 2012.

[108] Gísli R. Hjaltason and Hanan Samet. Properties of embedding methods for similarity searching in metric spaces. *IEEE Trans. Pattern Anal. Mach. Intell.*, 25(5):530–549, May 2003.

[109] Botao Peng, Themis Palpanas, and Panagiota Fatourou. Paris+: Data series indexing on multi-core architectures. *TKDE*, 2020.

[110] Themis Palpanas. Evolution of a Data Series Index. *CCIS*, 2020.

[111] Gustavo E. A. P. A. Batista, Xiaoyue Wang, and Eamonn J. Keogh. A Complexity-Invariant Distance Measure for Time Series. In *SDM*, pages 699–710. SIAM / Omnipress, 2011.

[112] Sam B Nadler et al. Multi-valued contraction mappings. *Pacific Journal of Mathematics*, 30(2):475–488, 1969.

[113] W. Johnson and J. Lindenstrauss. Extensions of Lipschitz Mappings into a Hilbert space. *Comtemporary Mathematics*, 26:189–206, 1984.

[114] Jiří Matoušek. Bi-lipschitz embeddings into low-dimensional euclidean spaces. *Commentationes Mathematicae Universitatis Carolinae*, 31(3):589–600, 1990.

[115] Nik Weaver. *Lipschitz algebras.* World Scientific, 1999.

[116] Nathan Linial, Eran London, and Yuri Rabinovich. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15(2):215–245, 1995.

[117] Kaspar Riesen and Horst Bunke. Graph classification by means of lipschitz embedding. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 39(6):1472–1483, 2009.

[118] Christos Faloutsos and King-Ip Lin. *FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets*, volume 24. ACM, 1995.

[119] Gabriela Hristescu and Martin Farach-Colton. Cluster-preserving embedding of proteins. Technical report, Technical Report 99-50, Computer Science Department, Rutgers University, 1999.

[120] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. Efficient similarity search in sequence databases. pages 69–84, 1993.

[121] Kin-Pong Chan and Ada Wai-Chee Fu. Efficient time series matching by wavelets. In *Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337)*, pages 126–133, Mar 1999.

[122] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016.

[123] H. Hotelling. Analysis of a complex of statistical variables with principal components. *Journal of Educational Psychology*, 24:417–441, 1933.

[124] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429, New York, NY, USA, 1994. ACM.

[125] G. Hua, M. Brown, and S. Winder. Discriminative learning of local image descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(01):43–57, jan 2011.

[126] Xin Dong, Lei Yu, Zhonghuo Wu, Yuxia Sun, Lingfeng Yuan, and Fangxi Zhang. A hybrid collaborative filtering model with deep structure for recommender systems. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[127] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3156–3164, 2015.

[128] Donald J Berndt and James Clifford. Using Dynamic Time Warping to Find Patterns in Time Series. In *AAAIWS*, pages 359–370, 1994.

[129] Gautam Das, Dimitrios Gunopulos, and Heikki Mannila. Finding similar time series. *Principles of Data Mining and Knowledge Discovery*, pages 88–100, 1997.

[130] Johannes Aßfalg, Hans-Peter Kriegel, Peer Kröger, Peter Kunath, Alexey Pryakhin, and Matthias Renz. Similarity Search on Time Series Based on Threshold Queries. In *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*, pages 276–294, 2006.

[131] Yueguo Chen, Mario A. Nascimento, Beng Chin Ooi, and Anthony K. H. Tung. SpADe: On Shape-based Pattern Detection in Streaming Time Series. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 786–795, 2007.

[132] Xiaoyue Wang, Abdullah Mueen, Hui Ding, Goce Trajcevski, Peter Scheuermann, and Eamonn Keogh. Experimental Comparison of Representation Methods and Distance Measures for Time Series Data. *Data Min. Knowl. Discov.*, 26(2):275–309, March 2013.

[133] Katsiaryna Mirylenka, Michele Dallachiesa, and Themis Palpanas. Data Series Similarity Using Correlation-Aware Measures. In *Proceedings of the 29th Interna-*

tional Conference on Scientific and Statistical Database Management, Chicago, IL, USA, June 27-29, 2017, pages 11:1–11:12, 2017.

[134] Hui Ding, Goce Trajcevski, Peter Scheuermann, Xiaoyue Wang, and Eamonn Keogh. Querying and mining of time series data: experimental comparison of representations and distance measures. *PVLDB*, 1(2):1542–1552, 2008.

[135] Kostas Zoumpatianos, Yin Lou, Themis Palpanas, and Johannes Gehrke. Query Workloads for Data Series Indexes. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, pages 1603–1612, 2015.

[136] Davood Rafiei. On Similarity-Based Queries for Time Series Data. In *Proceedings of the 15th International Conference on Data Engineering, Sydney, Austrialia, March 23-26, 1999*, pages 410–417, 1999.

[137] ESA. SENTINEL-2 Mission, 2018.

[138] Sloan Digital Sky Survey. `https://www.sdss3.org/dr10/data_access/volume.php`, 2018.

[139] ADHD-200. `http://fcon_1000.projects.nitrc.org/indi/adhd200/`, 2018.

[140] Incorporated Research Institutions for Seismology with Artificial Intelligence. Seismic Data Access. `http://ds.iris.edu/data/access/`, 2018.

[141] Michele Linardi and Themis Palpanas. Scalable, variable-length similarity search in data series: The ulisse approach. *PVLDB*, 11(13):2236–2248, 2018.

[142] Xavier Golay, Spyros Kollias, Gautier Stoll, Dieter Meier, Anton Valavanis, and Peter Boesiger. A new correlation-based fuzzy logic clustering algorithm for FMRI. *Magnetic Resonance in Medicine*, 40(2):249–260, 1998.

[143] Yoshihide Kakizawa, Robert H Shumway, and Masanobu Taniguchi. Discrimination and clustering for multivariate time series. *Journal of the American Statistical Association*, 93(441):328–340, 1998.

[144] Katarina Košmelj and Vladimir Batagelj. Cross-sectional approach for clustering time varying data. *Journal of Classification*, 7(1):99–109, 1990.

[145] Mahesh Kumar, Nitin R. Patel, and Jonathan Woo. Clustering seasonality patterns in the presence of errors. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 23-26, 2002, Edmonton, Alberta, Canada*, pages 557–563, 2002.

[146] Jin Shieh and Eamonn Keogh. iSAX: Indexing and Mining Terabyte Sized Time Series. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 623–631, New York, NY, USA, 2008. ACM.

[147] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An Optimal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions. *J. ACM*, 45(6):891–923, November 1998.

[148] Eamonn Keogh and Shruti Kasetty. On the Need for Time Series Data Mining Benchmarks: A Survey and Empirical Demonstration. *Data Min. Knowl. Discov.*, 7(4):349–371, October 2003.

[149] Kostas Zoumpatianos, Yin Lou, Ioana Ileana, Themis Palpanas, and Johannes Gehrke. Generating data series query workloads. *The VLDB Journal*, 27(6):823–846, December 2018.

[150] Eamonn Keogh and Chotirat Ann Ratanamahatana. Exact Indexing of Dynamic Time Warping. *Knowl. Inf. Syst.*, 7(3):358–386, March 2005.

[151] Davood Rafiei and Alberto Mendelzon. Similarity-based Queries for Time Series Data. *SIGMOD Rec.*, 26(2):13–25, June 1997.

[152] Davood Rafiei and Alberto O. Mendelzon. Efficient Retrieval of Similar Time Sequences Using DFT. *CoRR*, cs.DB/9809033, 1998.

[153] S. Albrecht, I. Cumming, and J. Dudas. The momentary Fourier transformation derived from recursive matrix transformations. In *Proceedings of 13th International Conference on Digital Signal Processing*, volume 1, pages 337–340 vol.1, Jul 1997.

[154] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases. *Knowledge and Information Systems*, 3(3):263–286, 2001.

[155] Kaushik Chakrabarti, Eamonn Keogh, Sharad Mehrotra, and Michael Pazzani. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. *ACM Trans. Database Syst.*, 27(2):188–228, June 2002.

[156] Jessica Lin, Eamonn J. Keogh, Stefano Lonardi, and Bill Yuan-chi Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery, DMKD 2003, San Diego, California, USA, June 13, 2003*, pages 2–11, 2003.

[157] Jin Shieh and Eamonn Keogh. iSAX: Indexing and Mining Terabyte Sized Time Series. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 623–631, New York, NY, USA, 2008. ACM.

[158] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th International Conference on*

Very Large Data Bases, VLDB '99, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[159] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized Product Quantization. *IEEE Trans. Pattern Anal. Mach. Intell.*, 36(4):744–755, April 2014.

[160] Marios Hadjieleftheriou. The libspatialindex API, January 2014. `http://libspatialindex.github.io/`.

[161] Paolo Ciaccia and Marco Patella. Bulk Loading the M-tree. pages 15–26, February 1998.

[162] Tolga Bozkaya and Meral Ozsoyoglu. Distance-based Indexing for High-dimensional Metric Spaces. *SIGMOD Rec.*, 26(2):357–368, June 1997.

[163] Claudio Maccone. Advantages of KarhunenLove transform over fast Fourier transform for planetary radar and space debris detection. *Acta Astronautica*, 60(8):775 – 779, 2007.

[164] Themis Palpanas. Evolution of a Data Series Index: the iSAX Family of Data Series Indexes. *Communications in Computer and Information Science (CCIS)*, "accepted for publication, 2020".

[165] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Zachary Zimmerman, Diego Furtado Silva, Abdullah Mueen, and Eamonn Keogh. Time series joins, motifs, discords and shapelets: a unifying view that exploits the matrix profile. *Data Mining and Knowledge Discovery*, pages 1–41, 2017.

[166] Southwest University. Southwest University Adult Lifespan Dataset (SALD). `http://fcon_1000.projects.nitrc.org/indi/retro/sald.html?utm_source=`

newsletter&utm_medium=email&utm_content=See%20Data&utm_campaign=indi-1, 2018.

[167] Skoltech Computer Vision. Deep billion-scale indexing. http://sites.skoltech.ru/compvision/noimi, 2018.

[168] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast Approximate Nearest Neighbor Search with the Navigating Spreading-out Graph. *PVLDB*, 12(5):461–474, 2019.

[169] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISAPP International Conference on Computer Vision Theory and Applications*, pages 331–340, 2009.

[170] Akhil Arora, Sakshi Sinha, Piyush Kumar, and Arnab Bhattacharya. HD-index: Pushing the Scalability-accuracy Boundary for Approximate kNN Search in High-dimensional Spaces. *PVLDB*, 11(8):906–919, 2018.

[171] William Johnson and Joram Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. In *Conference in modern analysis and probability (New Haven, Conn., 1982)*, volume 26 of *Contemporary Mathematics*, pages 189–206. American Mathematical Society, 1984.

[172] R. M. Gray and D. L. Neuhoff. Quantization. *IEEE Trans. Inf. Theor.*, 44(6):2325–2383, September 2006.

[173] Mohammad Norouzi and David J. Fleet. Cartesian K-Means. In *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '13, pages 3017–3024, 2013.

[174] Y. Kalantidis and Y. Avrithis. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2329–2336, June 2014.

[175] Yusuke Matsui, Yusuke Uchida, Hervé Jégou, and Shin'ichi Satoh. A Survey of Product Quantization. *ITE Transactions on Media Technology and Applications*, 6(1):2–10, 2018.

[176] Benjamin Bustos and Gonzalo Navarro. Probabilistic Proximity Searching Algorithms Based on Compact Partitions. *J. of Discrete Algorithms*, 2(1):115–134, March 2004.

[177] M. E. Houle and Jun Sakuma. Fast approximate similarity search in extremely high-dimensional data sets. In *21st International Conference on Data Engineering (ICDE'05)*, pages 619–630, April 2005.

[178] Edgar Chavez Gonzalez, Karina Figueroa, and Gonzalo Navarro. Effective Proximity Retrieval by Ordering Permutations. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(9):1647–1658, September 2008.

[179] Giuseppe Amato and Pasquale Savino. Approximate Similarity Search in Metric Spaces Using Inverted Files. In *Proceedings of the 3rd International Conference on Scalable Information Systems*, InfoScale '08, pages 28:1–28:10, 2008.

[180] Eric Sadit Tellez, Edgar Chávez, and Gonzalo Navarro. Succinct Nearest Neighbor Search. In *Proceedings of the Fourth International Conference on SImilarity Search and APplications*, SISAP '11, pages 33–40, New York, NY, USA, 2011. ACM.

[181] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegal. The Pyramid-technique: Towards Breaking the Curse of Dimensionality. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 142–153, New York, NY, USA, 1998. ACM.

[182] Beng Chin Ooi, Kian-Lee Tan, Kian-Lee Tan, Cui Yu, and Stephane Bressan. Indexing the Edges&Mdash;a Simple and Yet Efficient Approach to High-dimensional Indexing. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '00, pages 166–174, New York, NY, USA, 2000. ACM.

[183] Cui Yu, Beng Chin Ooi, Kian-Lee Tan, and H. V. Jagadish. Indexing the Distance: An Efficient Method to KNN Processing. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 421–430, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[184] C. Silpa-Anan and R. Hartley. Optimised KD-trees for fast image descriptor matching. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, June 2008.

[185] Gonzalo Navarro. Searching in Metric Spaces by Spatial Approximation. *The VLDB Journal*, 11(1):28–46, August 2002.

[186] Jon Kleinberg. The Small-world Phenomenon: An Algorithmic Perspective. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 163–170, New York, NY, USA, 2000. ACM.

[187] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. In *Similarity Search and Applications - 10th International Conference, SISAP 2017, Munich, Germany, October 4-6, 2017, Proceedings*, pages 34–49, 2017.

[188] Alexandr Andoni, Piotr Indyk, and Ilya P. Razenshteyn. Approximate Nearest Neighbor Search in High Dimensions. *CoRR*, abs/1806.09823, 2018.

[189] Yingfan Liu, Jiangtao Cui, Zi Huang, Hui Li, and Heng Tao Shen. SK-LSH: An Efficient Index Structure for Approximate Nearest Neighbor Search. *PVLDB*, 7:745–756, 2014.

[190] H. Shatkay and S. B. Zdonik. Approximate queries and representations for large data sequences. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 536–545, Feb 1996.

[191] Eamonn Keogh and Padhraic Smyth. A Probabilistic Approach to Fast Pattern Matching in Time Series Databases. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, KDD'97, pages 24–30. AAAI Press, 1997.

[192] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. Coconut: A Scalable Bottom-Up Approach for Building Data Series Indexes. *PVLDB*, 11(6):677–690, 2018.

[193] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. Coconut palm: Static and streaming data series exploration now in your palm. In *SIGMOD*, pages 1941–1944, 2019.

[194] L. Zhang, N. Alghamdi, M. Y. Eltabakh, and E. A. Rundensteiner. TARDIS: Distributed Indexing Framework for Big Time Series Data. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1202–1213, April 2019.

[195] Gisli R. Hjaltason and Hanan Samet. Ranking in Spatial Databases. In *Proceedings of the 4th International Symposium on Advances in Spatial Databases*, SSD '95, pages 83–95, Berlin, Heidelberg, 1995. Springer-Verlag.

[196] Stefan Berchtold, Christian Böhm, Daniel A. Keim, and Hans-Peter Kriegel. A Cost Model for Nearest Neighbor Search in High-dimensional Data Space. In *Proceedings*

of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS '97, pages 78–86, New York, NY, USA, 1997. ACM.

[197] Paolo Ciaccia, Marco Patella, and Pavel Zezula. A Cost Model for Similarity Queries in Metric Spaces. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, pages 59–68, New York, NY, USA, 1998. ACM.

[198] Paolo Ciaccia and Marco Patella. the power of distance distributions: Cost models and scheduling policies for quality-controlled similarity queries.

[199] Hnswlib - fast approximate nearest neighbor search. `https://github.com/nmslib/hnswlib`, 2019.

[200] DB Wang Group UNSW. SRS - Fast Approximate Nearest Neighbor Search in High Dimensional Euclidean Space With a Tiny Index. `https://github.com/DBWangGroupUNSW/SRS`, 2019.

[201] TEXMEX Research Team. Datasets for approximate nearest neighbor search. `http://corpus-texmex.irisa.fr/`, 2018.

[202] Yanping Chen, Eamonn Keogh, Bing Hu, Nurjahan Begum, Anthony Bagnall, Abdullah Mueen, and Gustavo Batista. The UCR Time Series Classification Archive, July 2015. `www.cs.ucr.edu/~eamonn/time_series_data/`.

[203] Andrew Turpin and Falk Scholer. User Performance Versus Precision Measures for Simple Search Tasks. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '06, pages 11–18, New York, NY, USA, 2006. ACM.

[204] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[205] Chris Buckley and Ellen M. Voorhees. Evaluating evaluation measure stability. In *SIGIR*, pages 33–40. ACM, 2000.

[206] Benito E Flores. A pragmatic view of accuracy measurement in forecasting. *Omega*, 14(2):93 – 98, 1986.

[207] Bahman Bahmani, Ashish Goel, and Rajendra Shinde. Efficient Distributed Locality Sensitive Hashing. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 2174–2178, New York, NY, USA, 2012. ACM.

[208] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *PVLDB*, 6(14):1930–1941, 2013.

[209] Anna Gogolou, Theophanis Tsandilas, Themis Palpanas, and Anastasia Bezerianos. Progressive Similarity Search on Time Series Data. In *Proceedings of the Workshops of the EDBT/ICDT 2019 Joint Conference, EDBT/ICDT 2019, Lisbon, Portugal, March 26, 2019.*, 2019.

[210] The Jemalloc library. `http://jemalloc.net/jemalloc.3.html`, 2020.

[211] Paris C. Kanellakis, Sridhar Ramaswamy, Darren E. Vengroff, and Jeffrey S. Vitter. Indexing for data models with constraints and classes (extended abstract). In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS 93, page 233243, New York, NY, USA, 1993. Association for Computing Machinery.

[212] Joseph M. Hellerstein, Elias Koutsoupias, and Christos H. Papadimitriou. On the analysis of indexing schemes. In *Proceedings of the Sixteenth ACM SIGACT-*

SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 97, page 249256, New York, NY, USA, 1997. Association for Computing Machinery.

SCALABLE AND ACCURATE HIGH-DIMENSIONAL SIMILARITY SEARCH: FROM DATA SERIES TO DEEP NETWORK EMBEDDINGS

Abstract: The world is drowning in a big data tsunami of high-dimensional objects that need to be analyzed in order to identify useful patterns and extract new knowledge in domains as varied as agriculture, medicine, cybersecurity, seismology, astrophysics, manufacturing, and finance, and others. In response to these needs, it is imperative to build analytical systems that truly support interactive exploration on datasets containing terabytes of high-dimensional objects, with dimensions reaching hundreds to thousands.

A fundamental and challenging operation called similarity search is the main bottleneck of many critical data processing tasks such as data cleaning, data integration and big data analytics (e.g., outlier detection, frequent pattern mining, clustering, and classification). A number of exact and approximate approaches have been proposed in the literature to support similarity search over massive data series collections.

In this thesis, we unify and formally define the terminology used for the different flavors of the similarity search problem. We present a similarity search taxonomy that classifies methods based on the quality guarantees they provide for the search results, and that unifies the varied nomenclature used in the literature. Following this taxonomy, we include a survey of similarity search approaches supporting exact and approximate search, bringing together works from the data series and multidimensional data research communities. We propose extensions to existing data series indexes that can answer approximate queries with guarantees and that outperform popular state-of-the-art techniques such as LSH, kNN graphs and quantization-based inverted indexes in many scenarios. We also design and conduct the two most exhaustive experimental evaluations in the field covering both exact and approximate techniques. Building upon the deep insights gained from both studies, we propose Hercules, a new algorithm that outperforms the state-of-the-art similarity search approaches in-memory and on-disk.

Our work has far-reaching fundamental and practical implications. We demonstrate that it is possible to design efficient high-dimensional vector similarity search algorithms with theoretical guarantees on the quality of the answers, and we thus offer a more promising alternative to the two current trends in the literature: (i) LSH-based algorithms that support guarantees, but are relatively slow, and (ii) kNN graphs and inverted indexes, which are relatively fast, but do not provide theoretical guarantees. This finding paves the way for very exciting new developments which will lead to efficient solutions that can support critical analytical tasks such as brain seizure detection, cyber-attack prevention, transportation management and data cleaning automation.

Keywords: Machine Learning, Data Mining, Similarity Search, Time Series, Data Series, Indexing, Query Processing