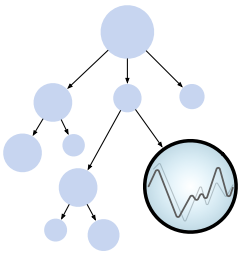

UNIVERSITÀ DEGLI STUDI DI TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE
**ICT International Doctoral School**

# INDEXING FOR VERY LARGE DATA SERIES COLLECTIONS

Konstantinos Zoumpatianos



Advisor:
Prof. Themis Palpanas
*Université Paris-Descartes*, France
*Università degli Studi di Trento*, Italy

Thesis Committee:
Prof. Gustavo Alonso, *ETH Zürich*, Switzerland
Prof. Christian Böhm, *Ludwig-Maximilians-Universität München*, Germany
Prof. Yannis Velegrakis, *Università degli Studi di Trento*, Italy

November 2016

# Abstract

*Data series are a prevalent data type that has attracted lots of interest in recent years. Specifically, there has been an explosive interest towards the analysis of large volumes of data series in many different domains. This is both in businesses (e.g., in mobile applications) and in sciences (e.g., in biology). In several time-critical scenarios, analysts need to be able to explore these data as soon as they become available, which is not currently possible for very large data series collections.*

*In this thesis, we present the first adaptive indexing mechanism, specifically tailored to solve the problem of indexing and querying very large data series collections. The main idea is that instead of building the complete index over the complete data set up-front and querying only later, we interactively and adaptively build parts of the index, only for the parts of the data on which the users pose queries. The contents and the resolution of the index are purely driven by query patterns; the more queries that arrive, the more data series are indexed and at a higher resolution. Adaptive indexing significantly outperforms previous solutions, gracefully handling large data series collections, reducing the data to query delay: by the time state-of-the-art indexing techniques finish indexing 1 billion data series (and before answering even a single query), our method has already answered* $3 * 10^5$ *queries. At the same time, we present novel algorithms for both full indexing of data series collections, as well as for efficient exact query answering. Our algorithms perform efficient skip-sequential scans of the data, avoiding the need of costly random accesses on the disk.*

*Moreover, up to this point very little attention has been paid to properly evaluating data series index structures, with most previous work relying solely on randomly selected data series to use as queries (with/without adding noise). In this thesis, we show that random workloads are inherently not suitable for the task at hand and we argue that there is a need for carefully generating a query workload. We define measures that capture the characteristics of queries, and we propose a method for generating workloads with the desired properties, that is, effectively evaluating and comparing data series summarizations and indexes. In our experimental evaluation, with carefully controlled query workloads, we shed light on key factors affecting the performance of nearest neighbor search in large data series collections.*

*Finally, apart from ad hoc data exploration, we also investigate methods for the systematic analysis of very large data series collections, supporting business intelligence applications. We present techniques, which borrow ideas from Strategic Management, for a goal-oriented anal-*

*ysis of large collections of performance indicator data series. Such algorithms can additionally be sped up through the use of the index structures presented in this work.*

**Keywords**[Data series, time series, indexing, adaptive indexing, data exploration, data mining, similarity search, nearest neighbors]

## Acknowledgements

With this thesis ends a chapter of my life that fundamentally changed the way I work, the way I think and the way in which I face problems that may arise both in a professional, as well as in a personal level. This period of study and personal development would have never been possible without the relentless and selfless support and guidance of my family, friends, colleagues, as well as all the people that I met during my journey. I want to express my deep gratitude to my parents for countless of reasons, for which there exists no space to list and no words to formulate.

I would like to thank my advisor, Prof. Themis Palpanas, whose guidance, company and wisdom made this journey a wonderful adventure towards knowledge. He was always there to support me, even before being asked. Additionally, I would like to thank Prof. Stratos Idreos, for his support, for sharing his deep knowledge with me and for the opportunities that he gave to me. This PhD would have never been possible without the endless meetings, discussions and support you and Themis so generously provided to me.

I would also like to express my gratitude to Prof. Johannes Gehrke from Cornell University, who hosted me and taught me important lessons on how research is done; Yin Lou, for sharing his brilliance and for giving solutions when nothing else worked; Prof. John Mylopoulos, for the opportunities that he gave me and the important lessons that he taught me; the colleagues with whom I worked together during the last years: Prof. Ioana Ileana, Michele Linardi, Dr. Alejandro Maté.

Last but not least, I want to thank all my lab-mates for being there when I needed a friend. Thank you dbTrento, DASLab and diNo!

With all my heart, I thank you all, and promise to help others as you helped me.

*Kostas Zoumpatianos*

---

*This research was partially funded by the FP7 EU ERC Advanced Investigator project "Lucretius" (grant agreement no. 267856) and by the FP7 EU IP project "KAP" (grant agreement no. 260111).*

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

*Up until the 1920s, everyone thought the universe was essentially static and unchanging in time. Then it was discovered that the universe was expanding. Distant galaxies were moving away from us. This meant they must have been closer together in the past. If we extrapolate back, we find we must have all been on top of each other about 15 billion years ago. This was the Big Bang, the beginning of the universe.*

Stephen Hawking

## 1.1 Data Series

In various scientific and industrial domains, it is often the case that analysts are required to measure quantities as they fluctuate over a dimension. These sets of fluctuating values are commonly called *data series* or *sequences*. The dimension over which data series are captured, stored and ordered depends on the application domain and can have various diverse physical meanings. Nevertheless, in every case, such sequences of recordings have to be captured, stored and analyzed as discrete objects rather than individual values. In this thesis, we concentrate on the problem of providing optimal access methods for very large collections of data series.

Data series (a.k.a. sequences) are present in virtually every scientific and social domain: they can be networking information (i.e., number of packets send per second), web usage data (number of requests per second etc.), environmental (weather, seismic, oceanographic data etc.), biological (DNA sequences, cardiograms, fMRI data, etc.), scientific (light-curves in astrophysics, mass-spectroscopy, etc.) as well as financial data (stock prices, sales, etc.) and industrial data (engine monitoring, power-

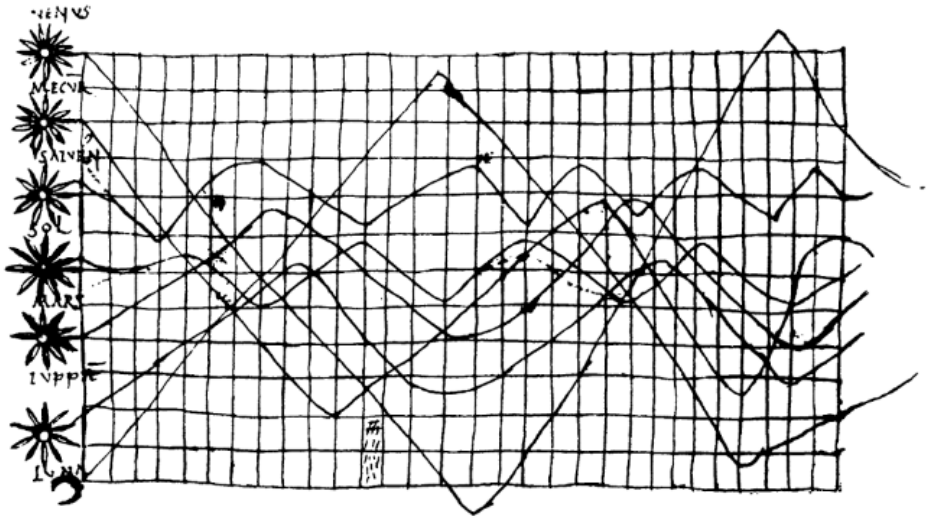


Figure 1.1: Time-series plot of planetary orbits from the late 10th century [60].

grid monitoring, etc.), to practically any kind of ordered set of numerical information [89, 187, 163, 140, 75].

**Definition 1 (Data Series or Sequence)** *Formally, a data series* $D = (d_1, ...d_n)$ *is defined as a sequence of points* $d_i = (v_i, p_i)$ *where each point is comprised of a value* $v_i$ *and a position* $p_i$ *associated to the order of this value in the sequence.*

Data series can be either streaming, where new values are continuously arriving, or static, where the length of each sequence is fixed. Further on, they can include uncertainty in the form of either a distribution or a set of distinct recordings for the same position. Depending on the application, data series can be ordered across any dimension. We continue our discussion with some of the most common dimensions over which data can be ordered and some application domains where such data types are frequently found.

### 1.1.1 Time-ordered Series (time-series)

By far, across most scientific disciplines and industrial domains, the most common dimension over which data are ordered is that of time. In this case we specifically talk about time-series. Scientists have used time-series to identify trends since hundreds of years ago. An interesting example can be seen in Figure 1.1, which is one of the earliest depictions of data graphed in the form of time-series, ever discovered. It comes from

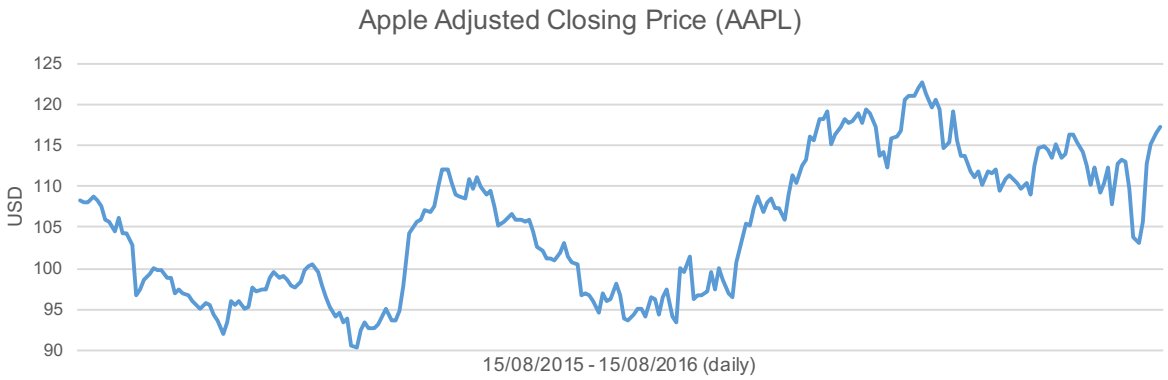



Figure 1.2: Example data series: Adjusted closing price of Apple Inc.

a late 10th century manuscript, discovered by Sigmund Günther in 1877, and it represents "a plot of the inclinations of the planetary orbits as a function of the time" [60]. A more contemporary example of time-series can be seen in Figure 1.2, depicting the closing price of Apple Inc.'s stock price as it fluctuates over time.

Through the years, there has been extensive research on time-series, both from a theoretical perspective (e.g. theory of stochastic processes, limit theorems etc.) as well as from a practical perspective of common statistics (e.g. regression, variance, etc.) [24]. Applications range from forecasting methods to correlation analysis, summarization, representation methods, sampling, outlier detection and more.

Such data can be commonly found in the following domains.

- In *biology* as electrocardiograms (ECG) and electroencephalograms (EEG).
- In *astrophysics* as orbits, rotation/cycles, events, light intensities, etc.
- In *environmental sciences* as weather related time-series (i.e., temperature, humidity, pressure, etc.).
- In the *internet-of-things* as distributed sensor readings.
- In *high energy physics* as particle fluxes, count rates, etc.
- In *marine science* as hydro-acoustic signals, seismic, gravimetric and magnetic measurements [115].
- In *finance* as stock market prices, product prices, as well as demands and volume of sales figures.

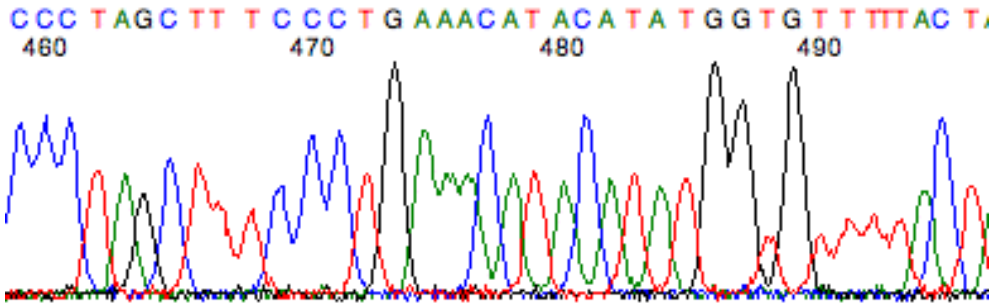



Figure 1.3: An example electrophoretogram (or trace) of a DNA sequence [4].

- In *industrial monitoring* as sensor readings from engines and industrial equipment, as well as power demand and consumption rates.
- In *social sciences* as various social indicators such as employment rates, education levels, crime rates and various other quantities, as they fluctuate over time.

### 1.1.2 Position-ordered Series

There are various applications that produce series ordered over position, with the most common being DNA sequences. In DNA data the precise order of each of the nucleotides within a DNA molecule is captured. The process for acquiring DNA sequences is by measuring the intensity of light emission at various bands. This is called an electrophoretogram trace and plots the intensity of each band at each position of the DNA. The peaks in this plot allow scientists and specialized software to identify the sequence of nucleotides. An example trace of a DNA sequence can be seen in Figure 1.3.

Another example of data ordered over position is word frequency counts in documents, which can be represented as sequences of frequencies. An example can be seen in Figure 1.4, where the frequency of the word "Agamemnon" is measured in three different versions of Homer's Iliad, available in Project Guttenberg. We can notice the interesting similarities in the trends among the three different books.

### 1.1.3 Mass-ordered Series

Mass-ordered series are commonly found in mass-spectroscopy. In this domain, scientists measure intensity of the ion signal over the mass-to-charge ratio. It is used to identify the structure of chemical compounds and it has applications in many diverse domains such as environmental sciences, forensics, clinical research, proteomics and

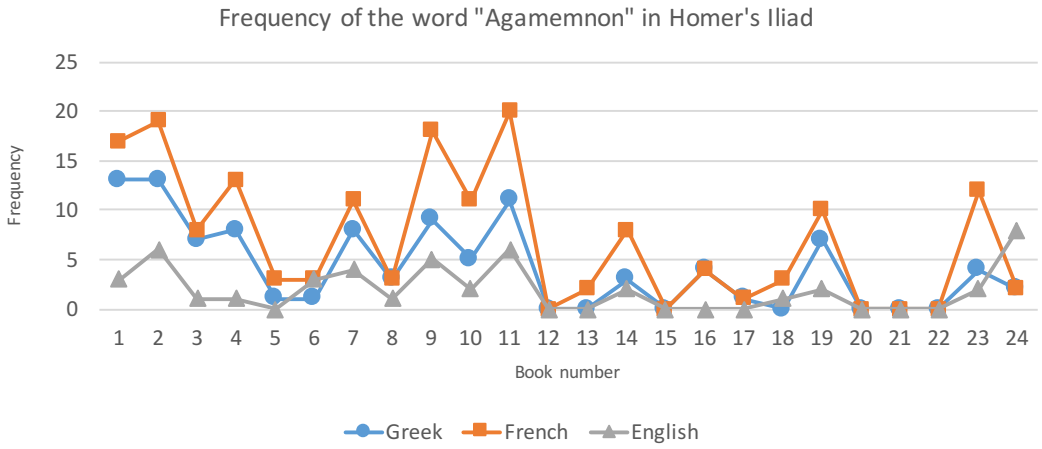



Figure 1.4: Frequency of the word "Agamemnon" as it fluctuates over books 1-24 of Homer's Iliad in the modern Greek, French and English translations.

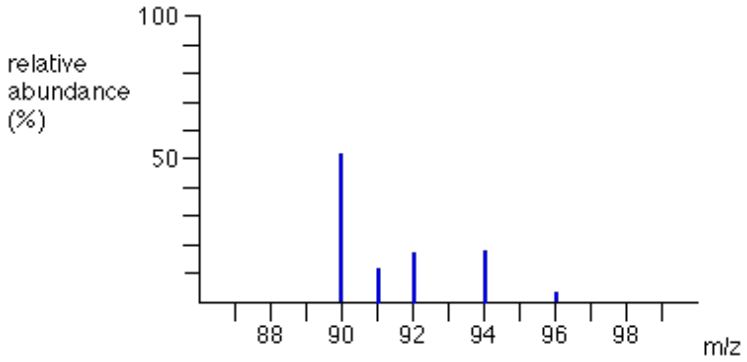



Figure 1.5: The mass spectrum of Zirconium (Zr) [2].

genomics. An example can be seen in Figure 1.5, where the mass spectrum of Zirconium is displayed.

### 1.1.4 Angle-ordered Series

Data ordered over angle are frequent in Astrophysics, where scientists analyze light curves at different viewing angles. Other cases where angle-ordered data are frequent are in domains where scientists need to find similar shapes [183]. For example, paleontologists need to identify similarities in the shapes of fossils and bones. In such cases, the outlines of the objects can be converted into sequences of values, by measuring the distance of each point of the contour to the center of the shape.

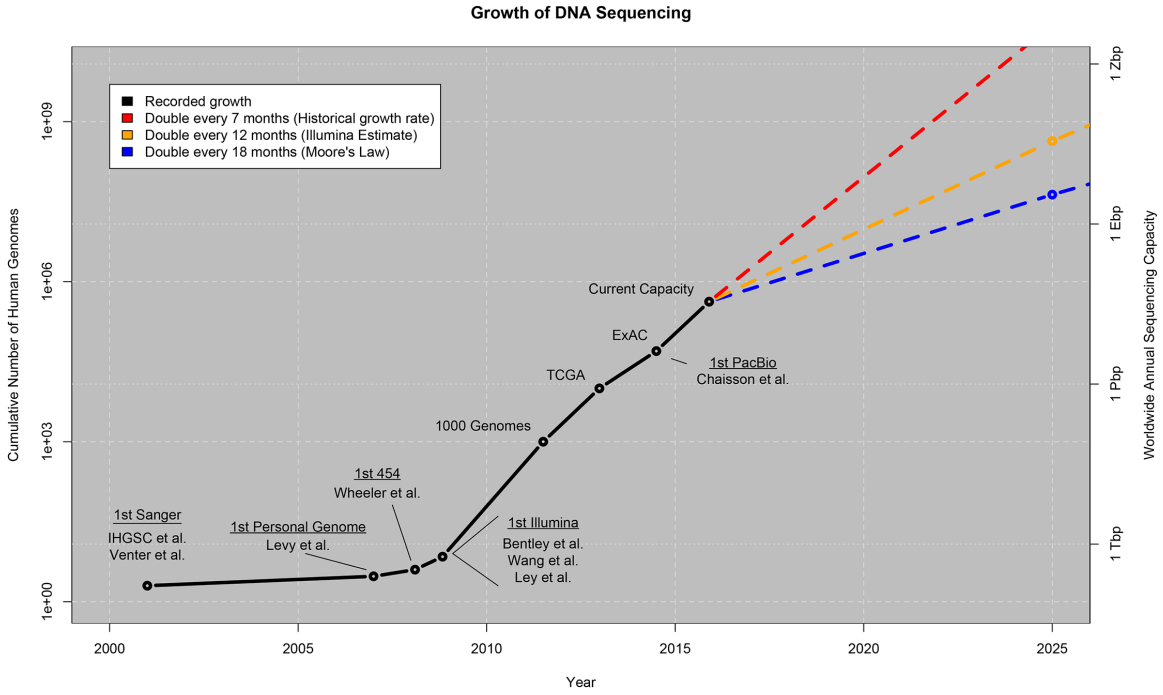



Figure 1.6: Growth of worldwide DNA sequencing in number of genomes sequenced (left) and in sequencing capacity (right) (taken from [169]).

## 1.2 Very Large Data Series Collections

Recent advances in sensing, networking, data processing and storage technologies have significantly eased the process of generating and collecting tremendous amounts of data series at extremely high rates and volumes, and very often the data volume is larger than the available storage space [122]. In addition, the processing cost for the data analysis is often bigger than the available processing power [172, 25]. It is not unusual for applications to involve numbers of sequences in the order of hundreds of millions to billions [5, 7]. As a result, analysts are unable to handle the vast amounts of data series that they have to filter and process.

Consider for instance that for several of their analysis tasks, neuroscientists are reducing each of their 3,000 point long sequences to just the global average, because they cannot handle the size of the full sequences [5]. Additionally, there are more than 70TB of spectroscopic sequence data from 200 million sky objects, collected by the Sloan Digital Sky Survey [7], allowing astronomers to study the universe. In biology, scientists are expected to collect around 2-40 ExaBytes of DNA sequence data by 2025 [169], as seen in Figure 1.6. In industry, there are various domains where huge data series col-

lections are produced. For example, each engine of a Boeing Jet generates 10 TeraBytes (TB) of data every 30 minutes [192]. Considering the length of flights and the amount of flights operating worldwide at any given day, the amount of data that are currently unexploited are in the order of exabytes.

All these data have to be processed and analyzed in their full detail, in order to identify patterns, gain insights, detect abnormalities, and extract useful knowledge. For this reason, there has been a significant interest in the data management community towards analyzing data series with the least possible processing and storage cost [134, 180, 26, 145, 50, 51].

## 1.3 Data Exploration Scenarios and Query Types

When confronted with very large data series collections, analysts need to run various monitoring, reporting or data mining applications. Simple monitoring and reporting applications usually involve simple Selection-Projection-Transformation (SPT) queries [122], which aim at filtering, transforming and presenting parts of the stored sequences. They are usually expressed as position/value constraints and transformation functions, and can be easily to implemented on existing database systems. On the other hand, complex data mining (DM) applications such as clustering, classification, deviation detection and more, involve expensive similarity search queries (nearest neighbor and range search), which require specialized algorithms and indexes in order to be efficiently processed.

### 1.3.1 Simple Selection-Projection-Transformation Queries

Selection-Projection-Transformation queries can be categorized under one or more of the following primitives:

- *Inter-sequence Filtering.* This primitive is used for filtering complete sequences that have specific properties at various positions. For example: "Bring all sequences for which the value went above X at least 5 times in a specified day."
- *Intra-sequence Filtering.* Queries in this category filter only a subset of the positions for all otherwise qualifying data series. An example is: "Bring the value for recordings taken between times 12:00 and 13:00 for all sequences".
- *Data Transformation.* Queries in this category apply a function on some positions of all valid sequences. Such transformations could manipulate one or multiple

sequences and can also perform aggregations across sequences or positions.

### 1.3.2 Complex Data Mining Queries

DM queries are more complex by nature: the processing has to take into consideration the entire sequence, which may involve thousands, or even millions of real-valued points. Examples are similarity queries, clustering, classification, outlier patterns, frequent sub-sequences, and more. These queries are not supported by current database systems, as they require specialized data structures, algorithms and storage methods to be efficiently performed.

Note that the data series datasets and queries may refer to either static, or streaming data. In the case of streaming data series, we are interested in the sub-sequences defined by a sliding window. The same is also true for static data series of very large size (e.g., an electroencephalogram, or a genome sequence), which we divide into sub-sequences using a sliding (or shifting window). The length of these sub-sequences corresponds to the patterns of interest. Because of their importance as the basis of most data mining algorithms, in this thesis we are interested in efficiently supporting similarity search queries.

## 1.4 Interactive Data Exploration

Firing exploratory queries, i.e., queries which are not known a priori, is becoming quickly a common scenario. That is, in many cases, analysts and scientists need to explore the data before they can figure out what the next query is, or even which experiment to perform next; the output of one query inspires the formulation of the next query, and drives the experimental process. In such cases, performing tuning and initialization actions up-front suffers from the fact that we do not have enough knowledge about which data parts are of interest [76, 80]. Such cases increasingly appear in modern scientific scenarios and in combination with the ability to collect large collections of data create a strong need for techniques that favor exploration. For example, the sequencing revolution in modern biology has given rise to the blooming fields of genomics and systems biology. High-throughput sequencing, a technique by which millions of DNA molecules can be read quickly and cheaply, turned the sequencing of a genome from a decade-long expensive venture to an affordable, commonplace laboratory procedure. Rather than painstakingly studying genes in isolation, we can now observe the behavior of a system of genes acting in cells as a whole, in hundreds

or thousands of different conditions. The sequencing revolution has just begun and a staggering amount of data has already been obtained, bringing with it much promise and hype for new therapeutics and diagnoses for human disease. For example, when a conventional cancer drug fails to work for a group of patients, the answer might lie in the genome of the patients, which might have a special property that prevents the drug from acting. This calls for exploration of the data series to find out the interesting patterns. With enough data comparing the relevant features of genomes from these cancer patients and the right control groups, custom-made drugs might be discovered, leading to a kind of "personalized medicine".

As data sizes grow even bigger, indexing data in a way that favors all possible query workloads means waiting for several days before even posing the first query. This can be a major show-stopper for many applications both in businesses and in sciences. For example, this is the case when high velocity financial tick data have to be processed in real-time for computing risks [6], or in vehicle monitoring, where jet airplane engine data need to be processed for early identification of potentially dangerous situations [146]. Similarly, in many applications, predefined queries are beneficial only if they can track data patterns or events within a given time limit; e.g., traffic monitoring applications for advertisement need to quickly determine user positions and interests.

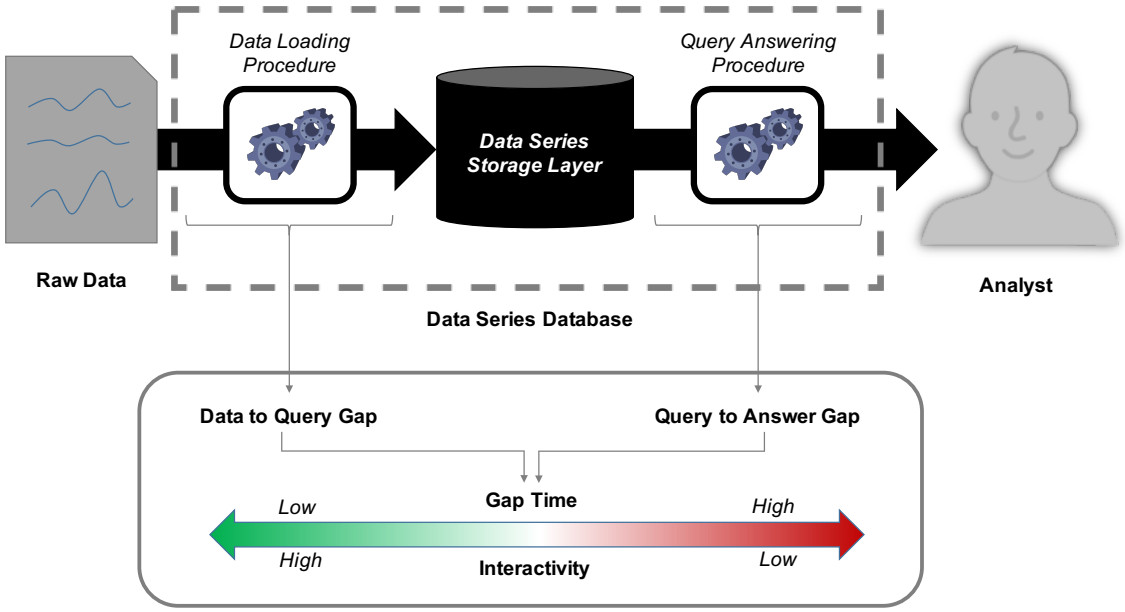



Figure 1.7: Data-to-Query time gap and Query-to-Answer time gap.

However, in order to provide quick response times for unknown workloads, data have to be loaded in specialized data systems. Such data systems use efficient data layouts and build indexes that allow for efficient query answering. However, constructing these index structures and moving the data to the appropriate layout requires a considerable amount of time to be spent, before even the first query is fired by the user. This is what is called the data to query gap. On the other hand side, the amount of time spent for answering the queries, given that the data are already loaded in such a system, is called the query to answer gap. The highest these gaps the less interactive a data exploration system is. A depiction of these gaps can be seen in Figure 1.7.

## 1.5 Contributions

In this thesis, we concentrate on the problem of providing efficient data structures and algorithms for enabling interactive exploration on very large data series collections. Our contributions can be broken down in five topics.

***Minimizing the data to query gap.*** We initially target the problem of minimizing the data to query gap, providing data structures that are specifically designed to minimize the time that an analyst needs to wait before being able to answer the first query. Our main idea is that instead of paying the cost of building the complete index over the complete data set up-front, we adaptively build the index only for the parts of the data that are related to the user queries. As a result, users can start answering queries faster, while the index building process is purely driven by query patterns; the more queries that arrive, the more data series are indexed and at a higher resolution. By the time state-of-the-art indexing techniques finish indexing 1 billion data series (and before answering even a single query), our adaptive index has already answered $3 * 10^5$ queries. Our work is described in detail in Chapter 3.

***Minimizing the query to answer gap.*** In the second part of this thesis, in Chapter 4, we describe methods for optimizing the query to answer gap in our data structures, allowing analysts to get immediate answers to their queries once the data have been loaded in the system. Our novel algorithms allow both full indexing, as well as efficient exact query answering. This is achieved by performing efficient skip-sequential scans of the data, avoiding the need of costly random accesses on the disk. In the case of full indexing, we perform a double pass over the raw data, keeping in memory a summarized version of the dataset. During the second pass, we move the raw data to the right locations on the disk. Our exact query answering algorithm, facilitates approximate search to get an initial answer. This answer is used to prune the data

(based on the lower bounds of their in-memory summaries), while they are scanned in a skip-sequential mode. Both our full indexing method and our exact query answering algorithm outperform the state-of-the-art and demonstrate that the most important bottleneck in data series indexing is random access.

***Interactive data series exploration.*** In Chapter 5, we present RINSE [196], a system that allows users to explore large collections of data series, using an interactive user interface. It facilitates our adaptive data series indexes and demonstrates their benefits in large scale data processing scenarios. Users can draw queries (data series) using a mouse, or touch screen, or they can select from a predefined list of data series. RINSE can scale to large data sizes, while drastically reducing the data to query delay.

***Generating workloads for data series indexes.*** Our fourth contribution is motivated by the fact that up to this point very little attention had been paid on how to properly evaluate data series index structures. Most previous work relied solely on randomly selecting data series with or without adding noise, which were then used as queries. A hardness analysis of these queries was always omitted, instead measuring index performance as the average query answering time across a large number of queries. On the contrary, in the context of relational databases, various benchmark workload generation techniques have been proposed through the years. Such techniques included methods for generating queries with specific properties, carefully designed to stress different parts of the database stack. In this thesis, we argue that apart from creating novel data structures for data series, there is also a need for carefully generating a query workload, such that these structures are stressed at appropriate levels. To solve this problem, in Chapter 6, we define measures that capture the characteristics of queries, and we propose a method for generating workloads with the desired properties, that is, effectively evaluating and comparing data series summarizations and indexes. In our experimental evaluation, with carefully controlled query workloads, we shed light on key factors affecting the performance of nearest neighbor search in large data series collections.

***Exploring meta-data enriched data series in a systematic way.*** In addition to all aforementioned contributions, which aim at speeding up and understanding ad hoc data series exploration, in Chapter 7, we provide a study on how to effectively do systematic data series exploration in Business Intelligence applications [201]. This has been a motivating scenario for our work, as real time business intelligence systems are in the need for interactive similarity search. Such operations allow algorithms to perform efficient clustering and deviation detection. As we will argue in this chapter, similarity search is crucial for identifying threats and opportunities that organizations can exploit

in a systematic way, which fits their strategic requirements [200].

### 1.5.1 Publications Produced

The work presented in this thesis has appeared in the following papers.

**Indexing for interactive data series exploration**

- K. Zoumpatianos, S. Idreos, and T. Palpanas. Indexing for interactive exploration of big data series. In *SIGMOD*, 2014
- K. Zoumpatianos, S. Idreos, and T. Palpanas. Rinse: Interactive data series exploration. In *VLDB*, 2015
- K. Zoumpatianos, S. Idreos, and T. Palpanas. Ads: the adaptive data series index. *The VLDB Journal (accepted for publication)*, 2016

**Generating workloads for data series indexes**

- K. Zoumpatianos, Y. Lou, T. Palpanas, and J. Gehrke. Query workloads for data series indexes. In *KDD*, 2015
- K. Zoumpatianos, Y. Lou, I. Ileana, T. Palpanas, and J. Gehrke. Generating data series query workloads. *Under Submission*, 2016

**Meta-data enriched data series exploration**

- K. Zoumpatianos, T. Palpanas, and J. Mylopoulos. Strategic management for real-time business intelligence. In *Enabling Real-Time Business Intelligence Workshop (BIRTE), held in conjunction with VLDB*, 2012
- K. Zoumpatianos, T. Palpanas, J. Mylopoulos, A. Maté, and J. Trujillo. Monitoring and diagnosing indicators for business analytics. In *CASCON*, 2013
- A. Maté, K. Zoumpatianos, T. Palpanas, J. Trujillo, J. Mylopoulos, and E. Koci. A systematic approach for dynamic targeted monitoring of kpis. In *CASCON*, 2014

# Chapter 2

# Background and Related Work

## 2.1 Data Series in Data Management Systems

Through the years, there has been a plethora of database systems, designed to handle data in the form of data series. Such systems have either been designed specifically, from top to bottom, to support sequential data, or they have been adapted to support them. We list a brief overview of both academic and commercial systems, their differences and evolution over time. A time-line per category for those systems can be seen in Figure 2.1.

### 2.1.1 Array Management Systems

One of the first systems used to analyze sequential data was the programming language APL developed during the 60s by Kenneth Iverson at Harvard University. It was a general purpose array manipulation language that operated on multidimensional arrays, residing in main memory. In 1988 and subsequently in 1992, the financial institution Morgan Stanley extended APL to A and A+ respectively [10]. These new data manipulation languages where suitable for large scale, high performance, financial data series processing. In 1993, A+ transformed into the successful commercial Financial Database Systems kdb and kdb+ by kx systems [155]. kdb+ is a proprietary disk and in-memory column-oriented database system for financial applications. It can be queried through Q and q-sql and allows for complex statistical operations to be efficiently performed on large amounts of financial data. In a separate effort, one of the first generalized large scale array processing systems, Rasdaman [16] was presented in 1997. It allowed for storage and retrieval of multi-dimensional array data on an Object-Oriented DBMS, using an extension of SQL called RasQL. A top to bottom

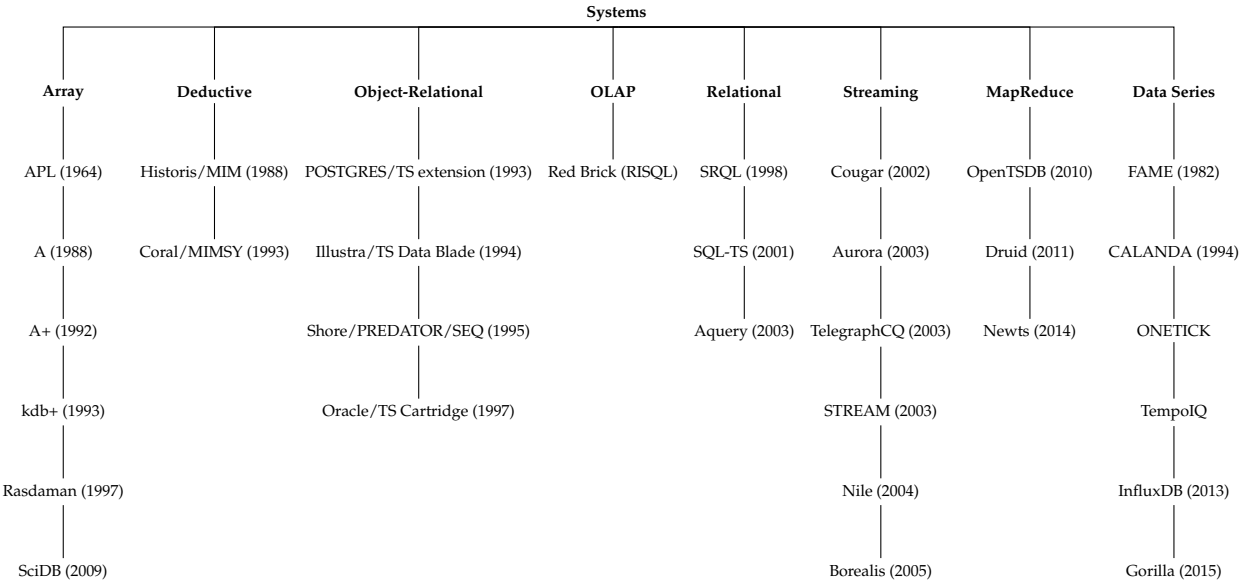



Figure 2.1: Time-line per category for systems that can support data series

array oriented database management system named SciDB was presented in 2009 [49].

### 2.1.2 Deductive Database Systems

Another type of database systems, which have been adapted to support data series, is that of deductive databases. Researchers working at the MCC laboratories, in the LDL deductive database system [46], have started Logical Information Machines in 1988. Its purpose was to provide scalable database technology for the analysis of financial time series. They developed a commercial database system, called Historis [108] (based on Linear Temporal Logic) and the MIM system above it. Historis stores data on disk in the form of compressed chunks and uses index structures in order to access them. It further on uses data structures such as simple arrays, hashes, nested arrays, B-Trees and can be disk based in full resolution and in-memory at lowest resolutions. It is heavily domain specific and can even be queried in almost natural language. It considers business related events when measuring time, such as specialized business calendars, holidays etc, for providing identifiers to records. In a separate effort, the Coral deductive database system [137], which was introduced in the early 90s by the University of Wisconsin-Madison, was adapted to offer similar to the MIM system, time series management functionality. This was implemented as an extension to Coral, through the MIMSY project [148] in 1993.

### 2.1.3 Object-Relational Database Systems

One of the first Object-Relational Database Systems was POSTGRES [171], introduced in 1986 at U.C. Berkeley. POSTGRES has been extended in 1993 to specifically support financial time series processing [40]. Time series were described as arrays with metadata such as frequency, life span etc. It supported various transformation operations on time series such as frequency change, mathematical operations and more. In the early 90s, POSTGRES served as the basis for the commercial database system: Illustra. Illustra, at the time, supported what was known as Data Blade modules [173]. Each Data Blade could introduce domain specific data types, access methods, and functions. The Time Series Data Blade [157], which was introduced in 1994, allowed Illustra to support both regular and irregular time series, perform mathematical operations on them, join different sequences based on their timestamps, and setup data feeds for streaming data ingestion. Illustra was later acquired by Informix, and Informix by IBM. In 1994, the object-relational Shore storage manager [32] was developed at University of Wisconsin-Madison. Above Shore, the SEQ system [133, 160, 159, 161] was developed in 1995, as a component of the PREDATOR database system [162]. SEQ was able to store and manipulate data series as a custom data type with specialized functions. Finally, Oracle had also developed a Time Series Cartridge in 1997 for its database system, which was later merged into Oracle's SQL Analytic Functions [34].

### 2.1.4 OLAP/Data Warehouses

Another domain where sequential data gathered attention, albeit low, was that of online analytical processing (OLAP). Specifically, the RED BRICK data warehouse [142, 143, 141] included support for time series query formulation in their RISQL language.

### 2.1.5 Data Series Extensions on Relational Databases

As it is expected, relational databases, by far the most popular database model, have also been extended to support data series processing. One of the first attempts to do so was SRQL [138], the Sorted Relational Query Language, introduced in 1998. It extends SQL with support for sorted relational data. In 2001, TS-SQL [149, 150] has been proposed, additionally supporting pattern search queries, and new techniques for optimizing such queries. Further on, the Aquery system [101], proposed in 2003, included an ordered relational model and algebra, as well as query language that supported sorted data from the ground up.

### 2.1.6 Streaming Database Systems

During the early 2000s, there has been an intense interest in developing systems able to cope with streaming data for sensor network applications [22]. Such data where most of the times in the form of streaming time-series. One of the earliest systems supporting querying on sensor data was Cougar [186], introduced in 2002. Other more mature examples of streaming databases included TelegraphCQ [41], Aurora [33] and Borealis [35], Nile [71], STREAM [116, 12]. In regards to commercial systems, there were various products developed during the same period. Those included: Streambase (later acquired by TIBCO), SQL Server StreamInsight, IBM InfoSphere Streams and more.

### 2.1.7 MapReduce Data Processing Systems

In the recent years, there has been an explosion on data systems that operate on distributed data using the map reduce paradigm. Such systems are either based in Hadoop, or developed as completely separate projects. OpenTSDB [121] is based on Hadoop and HBase and provides time series data processing functionality. Druid [55] (2011), which uses HDFS as the underlying storage system, provides an OLAP interface for analytics on very large collections of time series. Another system specifically designed for time series, called Newts [120] was introduced in 2014 and is based on Apache Cassandra.

### 2.1.8 Specialized Systems for Data Series

A last category of systems that are able to perform analytics on data series is those that have been specifically designed for this task and that do not fall in any of the previously defined database paradigms. One of the first time series management systems was developed for the finance sector, it was called FAME (Forecasting, Analysis and Modeling Environment) [1], and it was developed in 1982 by SunGuard (later acquired by Citigroup, and sold to private investors). FAME contains a database engine and analytical tools. It is structured as an object oriented database, where each time series is an object. These databases are stored in the file system. It additionally offers a 4GL language for plotting data and issuing queries. Another similar database was Calanda [53, 54], and it was a top to bottom database management system for managing financial time series. It was developed in 1995 by the Union Bank of Switzerland (UBS) and was later acquired by Ecofin. Finally, another database developed specifically for

financial time-series is ONETICK [3]. It is specialized for stock market data and offers multiple functions for domain specific data analytics, as well as a proprietary storage engine. Other systems specifically designed for time-series processing are TempoIQ, built for supporting IoT applications, Gorilla [129], which was developed by Facebook for monitoring their infrastructure, and InfluxDB [82], which is an open source time series managements system able to scale to very large data collections.

## 2.2 Data Series Exploration

Capturing trends, mining patterns, answering correlation and similarity queries are often among the requirements that analysts demand from data series exploration systems. It can be generally said that the main target of such systems is to enable analysts, or organizations in general, to achieve a set of goals or to discover new insights. These goals are formed by the expectations of an analyst from the data, and can be verified or dismissed. Insights on the other hand are unexpected information that might be derived from the data, which on their turn could introduce new theories or alter existing ones.

It is a job of a data mining system to assist the analyst to automatically or semi-automatically assess the degree to which his or her pre-specified theories about the data hold, as well as to propose different insights that were not present in them. These insights may form surprises that an analyst may be willing to inspect [154, 153] and explain [152]. This process can be offered in an ad-hoc way, where the analyst is issuing a set of exploratory queries, or in a goal-oriented, systematic way, where analysts are interested in automatically evaluating their formal expectations.

Such systems could operate both in raw data series, as well as in data series enriched with meta-data. Meta-data enriched data series are those that apart from the raw values of the data series, also contain attributes that describe them. For example time-series that record sales could have additional attributes that denote the location of the sales, or the demographic group to which they refer to. Such attributes can additionally form hierarchies and allow algorithms to aggregate data series over the various dimensions. On the other hand, raw data series do not include any additional information over which they can be categorized, and as a result are treated as bulk collections of sequences.

### 2.2.1 Meta-data Enriched Data Series

Traditionally, data enriched with multiple semantic dimensions of information, are stored in data warehouses. Data warehouses support decision making by providing On-Line Analytical Processing tools (OLAP) [48] for the interactive analysis of multidimensional data [83]. Such tools allow a person (analyst) to quickly acquire important information drilling in and out of the most interesting aggregates of a database. This kind of data have a set of dimension features and a set of measures. Dimension features are semantic facts about the data and they can form hierarchies. Example dimensions are countries, product types, categories and more. An example hierarchy could be in the country dimension which can include cities, regions and countries. Measures are the quantities that an analyst is interested on capturing over the different dimensions. Example measures are: sales, prices, counts and more. The common case for updates to a data warehouse is the Extract-Transform-Load (ETL) processing [175, 88, 130, 124], i.e., data are extracted from the sources and loaded to the data warehouse during specified time intervals.

Traditional data cube technology is good for static aggregates. However, it fails to readily explain trends over the time dimension [45]. For this reason, linear regression analysis in time series based data cubes has been presented in [72, 45]. Furthermore, the efficient generation of logistic regression data cubes was studied in [182], where the authors introduced an asymptotically loss-less compression representation (ALCR) technique, as well as an aggregation scheme on top of it, in order to eliminate the need of accessing the raw data for the calculation of each distinct cell in the data cube. With regards to the special case of reducing the size of a time series based OLAP data cube, a tilted time-frame scheme aiming at reducing the storage costs, as well as a popular-path based partial materialization of the multi-dimensional data cube for reducing the processing time are presented in [45]. Additionally, in [67] an extended Haar wavelet decomposition technique has been proposed for reducing the amount of data, as well as, for providing a reduced hierarchical based synopsis method for the time domain. In Section 2.3, we give a brief overview of methods related to management and analysis large collections of meta-data enriched data series.

### 2.2.2 Raw Data Series

In the previous section we talked about the importance of being able to manage data series organized in a multi-dimensional data cube. Apart from efficiently monitoring the hierarchies existing in a warehouse, we envision systems able to identify a wider

range of anomalies (e.g., anomalies not directly correlated to the existing hierarchies). In the case where such hierarchies are not available, groupings of related data segments could be hidden.

Towards this direction, identifying insights in very large collections of raw data series, across and beyond the multi-dimensional model, is not trivial. For this reason various algorithms and techniques have been developed. All these methods fall in the general category of Data Mining. According to Zaki and Meira [191], data mining is "the process of discovering insightful, interesting and novel patterns" in data, as well as being able to create predictive models from them.

There are various properties that make mining data series special [164]. One of them is their **extremely high dimensionality**. For example, if we consider each record of a sequence as a different dimension, then a data series of 2,000 points would correspond to 2,000 distinct dimensions. It is well known in data mining that treating high dimensional spaces is a hard problem [21]. Further on, data series **have a natural order in their values**. In this setting, we are not simply talking about high-dimensional vectors, but rather about sequences of recordings whose order is important. This property can be exploited by summarization algorithms, since dimensions that are close to each other tend to have similar values. As a result, raising opportunities for specialized data reduction algorithms to be developed. In Section 2.4, we give a brief overview of data series mining techniques, and their most important components, such as dimensionality reduction and similarity search.

## 2.3 Management and Analysis of Meta-data Enriched Data Series

Semantically enriched data can be stored in data warehouses. Such warehouses, allow analysts to perform analytics by aggregating data over various dimensions. In order to reduce the aggregation costs, Gray et al. [65] demonstrated the need of pre-calculating the possible Group-By combinations of the dimension attributes for all the measures. The resulting structure is called a Data Cube (Figure 2.2).

**Definition 2** *Formally a Data Cube $C$ is a multi dimensional cube, constructed over a set of dimensions $D = d_1, ..., d_n$, a set of measures $M = m_1, ..., m_k$ and a set of hierarchies over $H = h_1, ..., h_l$, where each hierarchy $h_i$ is composed by a set of dimensions $d_{h_i} \in D$.*

Finding the most informative parts of a data cube can be a fairly complex task, considering cases where there are a lot of dimensions [154] with multiple values. In

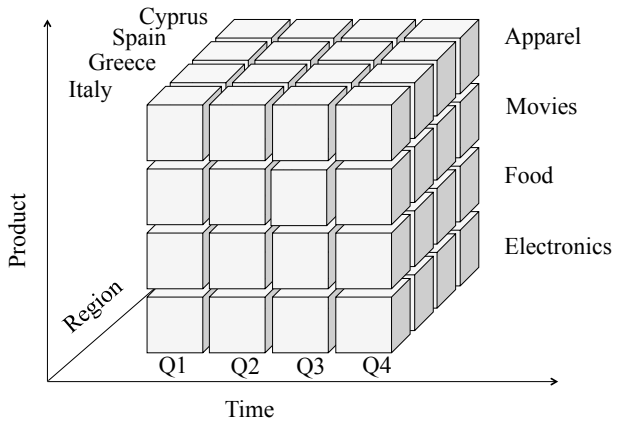



Figure 2.2: A sample Data Cube over Products, Countries and Time

such cases, it is crucial to assist and guide the analyst in an automated way, as the high-dimensionality of the search space makes the process of discovering exceptions [154] and explaining them [152] a daunting task. In the case where analysts have to deal with very large amounts of sequential data, simple aggregations may not be enough, as the dimension over which data are sequenced loses its significance. In such cases, it is important to be able to do regression, trend and pattern analysis on the complete sequences.

Traditionally data warehouses considered ordered dimensions the same way as all other dimensions in a data cube. Although, as Chaudhuri and Dayal noticed, dimensions such as time have a special significance [42]. To that end, there have been various techniques proposed for maintaining data series based OLAP cubes.

Specifically, the notion of time has been studied from the perspective of temporal databases, where it became known as temporal data warehousing [37, 112]. In such systems two distinct temporal information types are identified: the validity time and the transaction time. The validity time refers to the interval within which a record "holds", and the transaction time refers to temporal information (i.e., a point in time). For a detailed review the reader can look at [62]. Even though temporal data warehouses solved the problem of dimension and record evolution, they did not provide any means for monitoring trends and deviations in streams of data, and they also failed to capture the evolution dynamics of specific market-segments.

It becomes clear at this point that treating order as a common dimension can be rather limiting. This is especially clear in applications where predicting future incidents or the demonstration of complex behaviors over time, as well as their causes, is

important. Moreover, when this kind of temporal analytics have to be provided in a real-time and concise manner all over the data-set, data-warehouses have to treat the dimension of time, wherever it is found (validity time or transaction time), as a first class citizen and not solely rely on external tools for temporal data mining.

Various systems have been proposed in order to solve this problem, which adopt time-centric data representation techniques. These include the CHAOS system, which tackles the problem taking into account the business rules and their continuous evaluation against the data warehouse [67]. Other works have concentrated on creating OLAP architectures able to handle sequential data. Chen et al. [45] in 2002 presented a method for doing regression and trend analysis on top of a data cube of data series. Their work introduced the stream cube architecture in [72], with a goal of allowing for explanation of trends in the multi-dimensional space. Something that was not possible with traditional data warehouses, as they fail to capture trends.

For representing the data series, they used a tilted time frame data series representation, where time is split with varying granularity. This allowed them to use more information for the latest data and less for the older ones. They then fit a linear model on the minimal interesting level of aggregation detail in which a user is willing to analyze.

**Definition 3** *Formally a linear fit for a data series* $z(t) : t \in [t_b, t_e]$ *is a linear estimation function* $\hat{z}(t) = \hat{\theta} + \hat{\eta}t$.
Given the linear estimations $\hat{(z)}_i(t)$, of a set of cells $c_1, ...c_k$ and a cell $c_a$ which is calculated the aggregation of of the time-series in the cells $c_1$ to $c_k$, $z_a(t) = \sum_{i=1}^{K} z_i(t)$. It is proved that one can calculate $\hat{z}_a$ given just the linear estimations $\hat{(z)}_i(t)$ of its descendant cells.

Given this property, they are able to calculate all the cells based on the base cells. Additionally they presented a technique that only materializes the most popular paths of the Data Stream cube. Finally, in order to guide the analyst through the most interesting trends, by using exceptional regression lines, i.e. regression lines with a slope greater or equal than an exception threshold. They have implemented their methods for mining alarming incidents in their system called MAIDS [27].

In 2007, Li and Han [105] proposed a framework for identifying the top-*k* subspace anomalies in a multi-dimensional OLAP cube of time series. Using such a cube, one of the key challenges is that of identifying the different possible aggregate levels that are deviant. For example an aggregate level would be (Gender=M), where we get the sales to all male clients as a time series. We could also have an aggregate level of (Gender=F, City=Trento), where we would get the time series for all sales to women

in Trento. Using these aggregates we are now interested on guiding the analyst to the most deviant parts of the market. For example one could identify that women in Trento behave different than the women all over Italy, in regards to their shopping habits, and additionally to identify the trends of these differences.

Their system is able to create a multidimensional time-series cube, with the aggregates at all the possible hierarchy levels. At each level it contains the aggregated time series data of the most specific levels. For example in a database with Cities, Age-Groups and Genders, the aggregation of (City=Trento, Age-Group=18-25) would be defined as the linear aggregation of the time-series of the cells: (City=Trento, Age-Group=18-25, Gender=F), (City=Trento, Age-Group=18-25, Gender=M). Their observation is that similar subjects should behave similarly, for example men in Trento should behave "like all the men", and women in Trento should also behave "like all the women". Their technique uses the "top" aggregate levels (e.g. Genre="M") in order to calculate what the "expected" time series would look for the levels below it.

**Definition 4** *If p is a cell data-series cell in a Data Cube with data series $s_p$, and c one of its descendants with data-series $s_c$. Then as —c— is the the sum of all of the points of $s_c$ and —p— the sum of all the points of $s_p$. As an expected time-series for c the following is defined:*

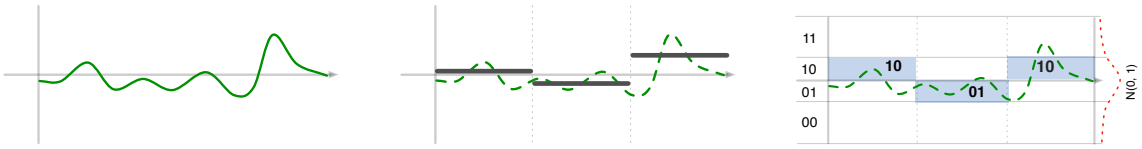
$$\hat{s}_c = \frac{|c|}{|p|} s_p$$

It intuitively means that the time-series $s_c$ should behave like the time series of the parent market segment $s_p$, but in proportion to their size. Based on comparing the expected time-series to the real time series they identify different types of outliers, group them and present the top outlying groups to the user as exceptions.

While all of them tackled the problem from various perspectives, to the best of our knowledge, there have been no comprehensive implementations that combine trend identification, as well as deviation detection query answering mechanisms, with the high-level information of the strategic objectives of a company. Additionally, there have been no tools that allow this kind of analytics to be done in a systematic way, allowing the automatic (and ad-hoc) generation and evaluation of such analysis queries.

## 2.4 Management and Analysis of Raw Data Series

There are various data mining procedures, which analysts can perform in large collections of data series. Those include tasks such as clustering [95, 181, 145, 135], classification, deviation detection [26, 39] and frequent pattern mining. One of the most




(a) A raw data series. (b) PAA representation of data series. (c) SAX representation of data series.

Figure 2.3: Example PAA/SAX representation for a data-series.

integral components of all these algorithms though, is that of finding similar data series in a database [11]. The query comes in the form of a data series $X$ and it says "find me the data series in the database which are most similar to $X$". Similarity is then measured using a given distance function. A common approach for answering such queries efficiently is to perform a dimensionality reduction technique and then use this representation for indexing. Lower bounding functions in the lower dimensionality spaces can be used to bound the true distances between data series, thus, allowing search algorithms to perform pruning. At the same time, a large set of indexing methods have been proposed for this kind of representations, including traditional multidimensional [68, 19] and specialized data series [165, 166, 30, 13, 180] indexes.

### 2.4.1 Summarizing Data Series

Since data series are inherently high-dimensional, different summarization techniques are used in order to reduce the total number of dimensions (it has been shown that the particular choice is of small importance [126, 125]). Popular techniques not only include well known transforms and decompositions such as Discrete Fourier Transforms (DFT) [131, 132, 103, 11], Discrete Haar Wavelets (DHWT) [103, 90, 38], Piecewise Constant Approximation (PCA), Adaptive Piecewise Constant Approximation (APCA) [36], Piecewise Linear Approximation (PLA) [92] and Singular Value Decomposition (SVD) [98, 139], but also data series specific data summarization techniques such as the Symbolic Aggregate approXimation (SAX) [107], Piecewise Aggregate Approximation (PAA) [93], and the indexable Symbolic Aggregate approXimation (*i*SAX) [165, 30]. We briefly describe the most prominent ones below.

**Piecewise Aggregate Approximation (PAA)**. In 2000, Yi and Faloutsos [188], as well as Keogh et al. [93], presented the idea of segmented means [188] or Piecewise Aggregate Approximation (PAA) representation [93]. This representation allows for

dimensionality reduction in the time domain, by segmenting the data series in equal parts and calculating the average value for each segment. An example of PAA representations can be seen in Figure 2.3; in this case the original data series is divided into 3 equal parts.

**Discrete Fourier Transform (DFT)**[1] **[131, 132, 103, 11]** uses Fourier transforms to convert a data series to the frequency domain and represents it as a list of coefficients. Fourier transforms are linear transforms that can convert a series of values into a number of sinusoids. A direct implication of the well-known Parseval's theorem is that Euclidean Distance between the two series in the frequency domain is exactly the same as in the original domain. As a result, a subset of the co-efficients can be used to estimate a lower bound for the real distance. It is important to note that in the frequency domain, the number of co-efficients is equal to the number of points of the original sequence, while each one of them (except the first) is a complex number. From a memory complexity perspective, this means that almost twice the space is required for storing the co-efficients over the raw data. However, as it has been shown in [132], the co-efficients have a symmetric property with respect to the middle. For this reason only half of them need to be stored and the others can be inferred. Additionally, when the data series are normalized (to have a mean value of zero, and a standard deviation of one), the first co-efficient is always zero [132]. Thus, it doesn't have to be explicitly stored. In regards to complexity, FFT can be computed in time $O(nlogn)$, and as a result it is not as efficient as its competitors. However, a novel line of works on Sparse FFT [73], is able to compute a subset $k$ of the FFT co-efficients (the ones with the highest energy) in $O(logn\sqrt{nklogn})$.

**Discrete Haar Wavelet Transform (DHWT) [103, 90, 38]** uses Haar wavelets in order to transform a data series into a list of coefficients. Much like the Fourier Transform, wavelets allow for the decomposition of a series using a function. Haar wavelets use a square-shaped function and can be efficiently computed in O(n) time.

**Symbolic Aggregate approXimation (SAX)**. Based on PAA, Lin et al. [106] introduced the Symbolic Aggregate approXimation (SAX) representation. It works by partitioning the value space in segments of sizes that follow the normal distribution. Each PAA value can then be represented by a character (or a small number of bits) that cor-

[1] In this work, we use the well known FFT algorithm.

responds to the segment that it falls into. This leads to a representation with a very small memory footprint, an important requirement for managing very large data series collections. A segmentation of size 3 can be seen in Figure 2.3, where the data series is represented with the SAX word "10 01 10".

**Piecewise Linear Approximation (PLA).** This family of algorithms approximates data series using directed linear segments. They have been used since the 1960s [29] and they are still used by data series summarization algorithms today [126]. They work by identifying the set of longest running lines, which can describe a set of points with an error less than a threshold. The collection of lines is used to reconstruct the signal. Error can be measured using various distance measures.

### 2.4.2 Data Series Similarity Search

Given an input data series, similarity search refers to the methods that are able to answer the following types of queries.

- **Nearest neighbor queries**, which return the top k nearest to the input data series in the dataset.
- **Range queries**, which return the data series that are within a predefined range $\epsilon$ of the input data series.

Data series similarity search methods fall in two broad categories. In the first category there are methods that are based on serial scan. In this case, data are scanned without any additional index structure to guide the search. In the second category, generic or specialized indexes can be used. In this case data are summarized using a summarization method and an index structure is built above the summaries. The index is then used to answer queries. In the rest of this section, we list works that fall in both these categories, and additionally give a brief description of some of the most important distance measures frequently used for mining data series.

#### Distance Measures

There are various distance measures that have been proposed for measuring similarity between different data series.

**Definition 5 (Distance Measure)** *Formally, a distance measure is a function* $d(x, y)$ : $R^n \times R^n \rightarrow R$, *which measures the distance between two data series* $x$, $y$.

When a set of properties hold, a distance measure is said to be a metric. One of those is the triangle inequality, which can be used to prune the search space. This property is exploited by metric indexes [190, 47] for performing efficient similarity search.

**Definition 6 (Metric Distance Measure)** *When the following properties hold, a distance function d, is said to be a metric.*

- $d(x, y) \geq 0$ *(non-negativity)*
- $d(x, y) = 0 \iff x = y$
- $d(x, y) = d(y, x)(symmetry)$
- $d(x, z) \leq d(x, y) + d(y, z)$ *(triangle inequality)*

Further on, in order to use data series summarizations and make exact query answering feasible, indexes use lower and upper bounds of the distances between two data series in the original data space. These bounds are computed based on the summarizations of the data series. Throughout our study, we refer to the lower bounding function of a given summary as function $L$.

**Definition 7 (Lower Bounding Function)** *Formally, given two data series $x$, $y$, the lower bounding function of their summaries $L(x, y) : R^n \times R^n \rightarrow R$ returns a lower bound of their true distance in the original space, such that: $L(x, y) \leq D(x, y) \; \forall \; x, y$.*

We hereby list a few of the most important distance measures used in literature.
**Euclidean Distance (ED)**. The most commonly used distance measure, usually chosen for its simplicity, is the Euclidean Distance (ED) [11], alternatively called the $L^2$ distance. It is the square root of the squared sum of the pair-wise point distances, and for two data series $x$,$y \in R^n$, it can be computed as follows:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{2.1}$$

It is generally known that Euclidean Distance on z-normalized data (mean 0, standard deviation 1) is equivalent to Pearson's Correlation [20]. However, this distance measure is unable to capture shifts in time or scaling differences across data series.

**Dynamic Time Warping (DTW)**. DTW [176, 134] is a dynamic programming algorithm rather than a distance measure itself. It is able to find the optimal match between points

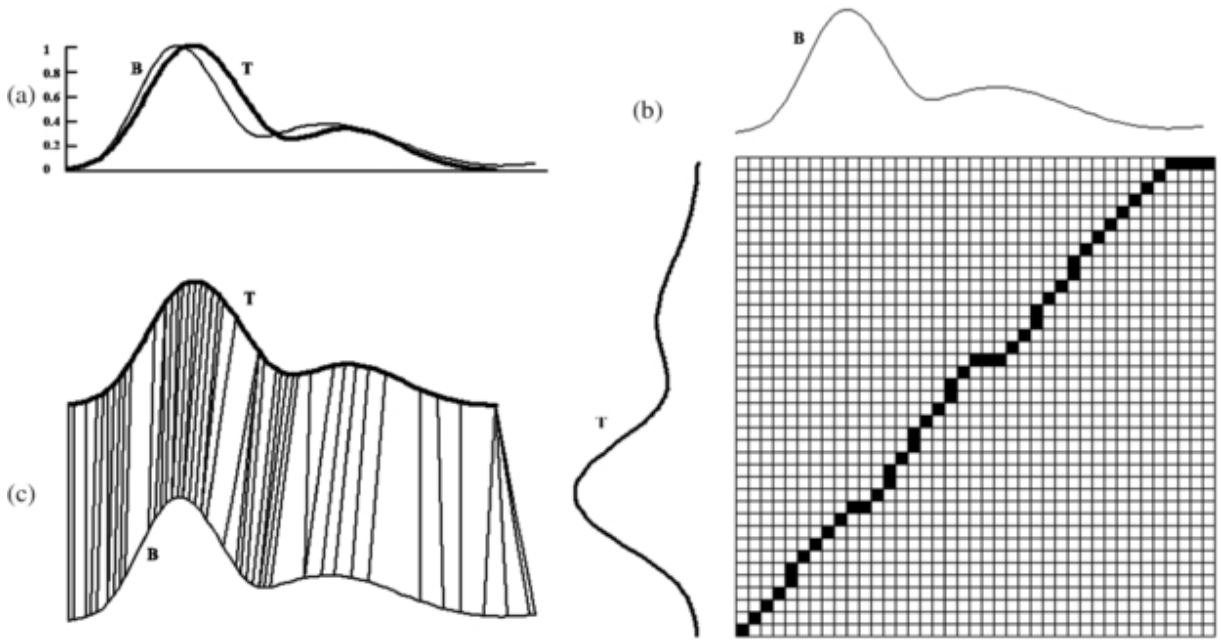



Figure 2.4: An example of the DTW algorithm. Taken from [104].

of different data series, thus allowing for local time shifts and variances in speed to be considered. It internally facilitates a user defined distance measure, most commonly the Euclidean Distance, to align points and find the optimal warping path. An example of the DTW algorithm can be seen in Figure 2.4, where two data series that are slightly shifted are displayed. The algorithm is able to align matching points and find the optimal warping path in the pair-wise distances matrix, as seen in Figure 2.4c.

**Longest Common Subsequence (LCSS).** LCSS [177] is a variation of the Edit Distance [102]. It matches data series with support for stretching, while allowing some of their points to remain unmatched. The idea intuitively is to use a threshold $\epsilon$, and two points are considered to match if their distance is less than this value.

**Edit Distance on Real Sequences (EDR).** Another distance measure, which is more robust to noise, shifts and scaling changes is Edit Distance on Real sequences (EDR) [43]. This measure is based on Edit Distance (ED) [102] and essentially counts the number of edit operations required to convert one data series to the other. Such edit operations are insert, delete and replace. Two points are considered equal if their $L^1$ distance is less than a user defined threshold $\epsilon$. Further on, EDR allows for the two data series to have different sizes.

#### Serial Scan-based Similarity Search

One of the simplest ways to evaluate similarity search queries, is that of performing a serial scan over the data series. Data series can either be in their raw form, or transformed using wavelets or Fourier transformations. This last category relies on carefully organizing and storing the data series representations on disk. Such techniques could also be considered as indexes, as data have to be stored in a structured way. Using this approach, data can be read in a step-wise function, where distance estimations for all data series are gradually refined as we read the summarizations in increasing detail. This technique has been first introduced in HierarchyScan [103], where data series were transformed into Fourier co-efficients. They were then sorted based on their discriminative power (i.e., their energy) and scanned on that order. This system was able to answer range similarity queries in that way. This is because of the monotonic property of Euclidean Distance, which also holds in the frequency space. When a candidate data series' distance increases above the similarity range threshold $\epsilon$, this data series can be pruned. As a result, this technique is able to avoid a full scan of the raw data file for answering a query. A similar technique, but for DHWT, has been proposed by [90]. In that case, nearest neighbor queries are additionally supported, through the introduction of novel upper and lower bounds for the distances of the transformed data. DWT co-efficients are read in a step-wise fashion and candidates can be dismissed based on these bounds. In another research direction, there are recent studies that have shown that in certain cases sequential scans on the raw data themselves can also be performed very efficiently [134]. Such techniques though, are only applicable when the database consists of a single, long data series, and queries are looking for potential matches in small subsequences of this long data series.

#### Index-based Similarity Search

As we've seen, nearest neighbor search can be an intensive task. The naïve approach requires a full scan of the dataset, while smarter scan approaches are able to re-organize data in such a way that lower bounding functions on summarizations make it possible to significantly prune the search space.

Apart from re-organizing data, index structures can also be build to speed up similarity search. Such data structures hierarchically organize data series in one or many levels of aggregation. At each level multiple groups of data series are summarized under a common representation. This is illustrated in Figure 2.5. In this figure, each node represents a summarization of all the data series below it, while the leaf level nodes

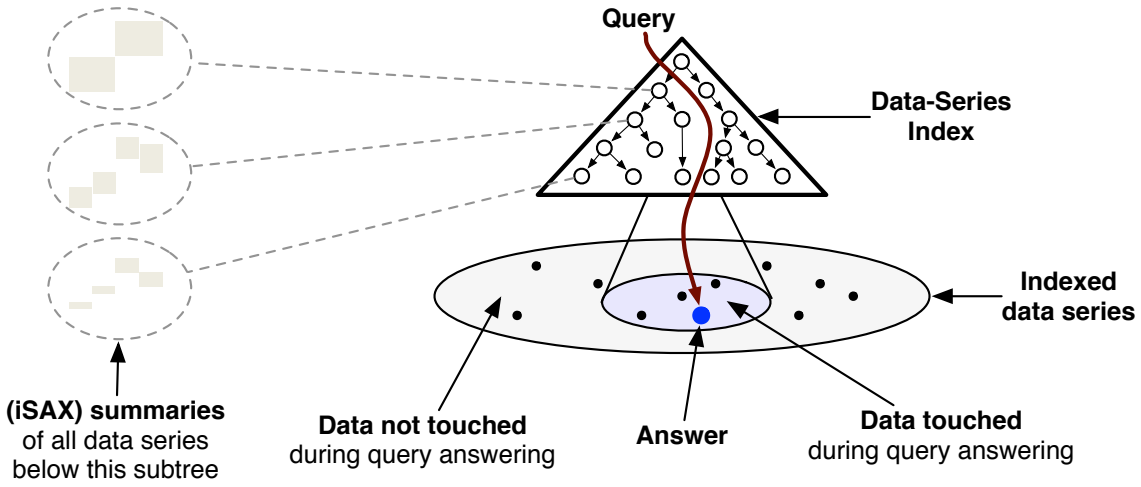



Figure 2.5: An index structure built above a set of data series, pruning the search space for a query.

correspond to a set of raw data series, usually stored on the disk. By computing lower bounds at every level of summarization (internal node of the tree), one can perform pruning. The higher in the tree the pruning, the more the data that are pruned are. Because we operate on the summaries, we need to dismiss false positives. For this reason, the raw data of the surviving leaves have to be read in full detail from the disk, in a last step, in order to guarantee exact answers. Such hierarchical data series indexes, supporting exact nearest neighbor search can be broadly divided in two categories as follows.

**Summarization & spatial access method**. The first category involves the use of a summarization technique and a (general) spatial access method. Data are summarized, and only the summaries are used for constructing a spatial index. In the leaf level the raw data series are stored in full detail, in order to guarantee correct answers. Previous work has proposed the use of R-Trees with summarizations like DFT [11, 59, 131, 132], DHWT [38] as well as Piecewise Linear Approximation (PLA) [44].

**Data series specific summarization & index**. The second category involves the use of a summarization method specific to data series, and a specialized index that is built on top of it. Such indexes include TS-Tree [13] (based on a symbolic summarization), DS-Tree [180] (based on APCA), ADS [195] and *i*SAX index [165, 30, 31] (built on an indexable version of SAX), and SFA index [156] (it uses a symbolic summarization of

data series in the frequency domain based on DFT).

In this work we will concentrate on the SAX representation, which can be extended to an indexable SAX (iSAX) [165]. This indexing method has been shown to scale very efficiently to very large dataset sizes [30, 31]. iSAX considers variable cardinality for each character of a SAX representation, and as a result variable degrees of precision. An iSAX representation is composed of a set of characters that form a word. Each word represents a data series available in the dataset. Each character in a word is accompanied by a number that denotes its cardinality (the number of bits that describe this character). In the case of a binary alphabet, with a word size of 3 characters and a maximum cardinality of 2 bits, we could have a set of data series (two in the following example) represented with the following words: $00_2 10_2 01_2$, $00_2 11_2 01_2$, where each character has a full cardinality of 2 bits and each word corresponds to one data series. If we now reduce the cardinality of the second character in each word, we could represent both of them with a single iSAX representation: $00_2 1_1 01_2$. That is because $1_1$ corresponds to both 10 and 11, since the last bit is trailed when the cardinality is reduced. By starting with a cardinality of 1 for each character in the root node and by gradually performing splits by increasing the cardinality by one character at a time, one can build a tree index [165, 166]. Such cardinality reductions can be efficiently calculated with bit mask operations.

The state-of-the-art iSAX 2.0 index is also based on this property [30, 31]; it is a data series index that implements fast bulk loading. Figure 2.6 depicts an example where each iSAX word has 3 segments and each segment a maximum cardinality of 4 (2 bits). The root node has $2^w$ children ($2^3$ in Figure 2.6) while each child node forms a binary sub-tree. Each leaf node corresponds to a split in one dimension and points to a single area of the domain.

A typical data series index, such as iSAX, contains both the summarized representations and the actual, raw data series values. The representations are used as index keys to efficiently guide index creation, as well as for answering similarity search queries by pruning the search space, i.e., eliminating candidate data series that cannot possibly be part of the answer (true negatives). The actual data series are also needed in order to eliminate the false positives, and produce the exact, correct answer.

Our contributions build on top of this line of work by enabling adaptive indexing using the state-of-the-art iSAX representations. Contrary to past work, our new adaptive index allows for incremental, continuous and adaptive index creation during query time. Initialization cost is kept low, bringing the ability to query the data set much sooner than in past work. We show both the significant bottleneck faced by

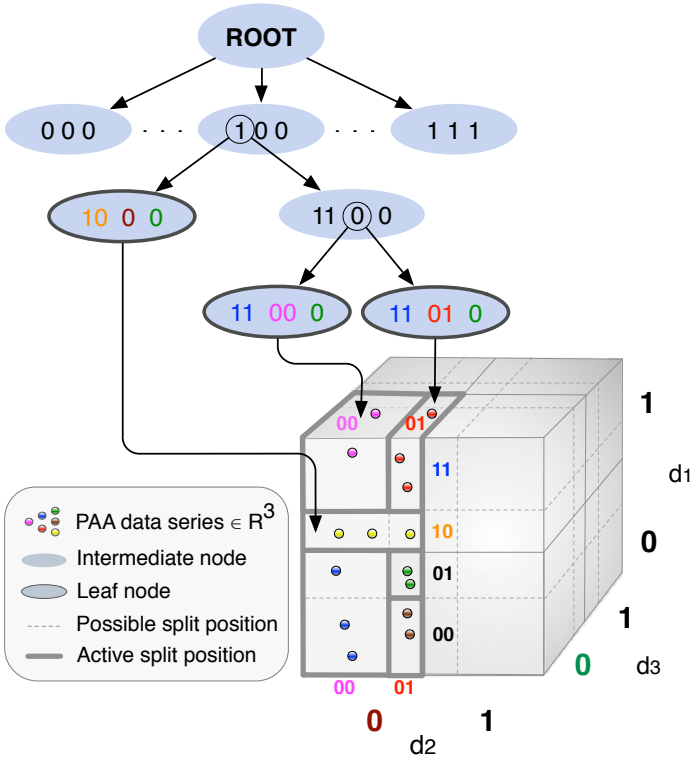



Figure 2.6: An example of iSAX and its space partitioning.

state-of-the-art indexing as we grow to large data, as well as the drastic improvement that adaptive indexing brings.

#### Evaluating Similarity Search

Although the problem of data series indexing has attracted considerable attention, there is little research so far in properly evaluating those indexes. There are two properties that are important when evaluating an index structure. One is the construction efficiency; i.e., the amount of time needed in order to construct an index. And the other is the time that an index requires in order to answer a query workload. Evaluating the first part is trivial, as it is simply a matter of measuring the time required in order to construct the data structure. However, evaluating query answering is not so easy, as there are various intrinsic query properties that affect the performance of an index. Nevertheless, most of the research has relied on rather simple solutions, such as either selecting a subset of the dataset as queries [30, 195], or on generating queries by adding small amounts of noise to existing data series [59, 11]. Nevertheless, there

is no work done on understanding the characteristics of the queries and their implications on query answering. In this thesis, we also focus on studying the properties of data series query workloads. The aim is to better understand the characteristics of different queries, and how these can be used to effectively test a data series index under different, but controllable conditions.

### 2.4.3 Data Mining Algorithms

Up to this point we have seen different distance measures, as well as methods for summarizing data series, such that lower bounds can be computed and used by index structures, in order to be able to perform efficient similarity search. All these techniques can be used in order to support efficient data mining on very large collections of data series. In this subsection we survey the most important work on data series mining algorithms.

#### Clustering

Clustering is the process which separates a dataset in groups. The data series which belong in each group should be very similar to one another, but data series belonging to different groups should be dissimilar. Clustering in data series can refer to clustering whole sequences or subsequences, in static or streaming data. Further on, different approaches can either operate on the raw data series, features extracted from the data series, or model parameters extracted from the data series. There are various generic clustering algorithms presented in literature that can be applied to clustering data series. Such methods can be split in multiple categories. However, the categories applied to data series settings are the following [127].

**Partitioning.** These methods partition a dataset in k clusters, where each cluster is represented by a single representative point. It is common to use the centroid (or mean). Such algorithms are the K-means [109] and its variants.

**Hierarchical.** The goal of hierarchical clustering is to split a dataset in a hierarchy of clusters, which form a tree structure. There are two ways to perform hierarchical clustering [191], agglomerative and divisive. Agglomerative methods start bottom up, creating clusters which then they gradually group to create a hierarchical structure. On the other hand, divisive methods start with one cluster, dividing it at each step of the process, in order to create the hierarchical structure.

**Spectral.** Spectral clustering also reduces the dimensions of the similarity matrix, before clustering. This not only reduces the number of dimensions, but also allows for

clusters to be found based on their connectivity and not strictly within convex boundaries. An example is two clusters in a two-dimensional space, forming concentric rings, where the outer ring is one cluster and the inner ring is another cluster.

#### Classification

Classification algorithms seek to assign a label in an unlabeled data series. They are trained using a dataset of labeled data series. Examples include the nearest neighbor classifier, which uses the k nearest neighbors to decide the label of an unlabeled data series. Decision trees that divide the dataset in a tree structure which is then followed in order to assign a label in an unseen example. Naive Bayes methods, which train a statistical model on the data. Neural networks that learn a function from the raw data space to the class label space, as well as support vector machines that create linear models for differentiating each class.

#### Deviation Detection

Given a model of normal behavior, deviation detection is the process of identifying anomalous data series. There are various ways of defining an anomaly or a deviation. Anomalies can be both within a data series itself, or across multiple data series. As a result, we can either talk about deviating records of a data series, i.e., behavior that is different from its previous points, or about deviating data series, which are sequences that are unexpected in regards to the rest of the dataset. This is a hard problem as there is no strict definition of an anomaly, and it can radically differ from one application to the other [136]. However, a general description of anomaly has been given in [94], where anomalous data series are defined as the ones that are the most different from all other data series. In this case as well, similarity search is very important in order to identify when a data series is similar to others.

#### Frequent Pattern Mining

Frequent patterns are subsequences that appear frequently in a data series database. They are alternatively called "motifs". In [185], the authors introduce an algorithm that is able to identify such patterns with invariance to uniform scaling. Further on, an algorithm for efficiently identifying motifs has been introduced in [118]. While a method that is able to scale to dataset sizes that don't fit in main memory has been presented in [117].

**More Topics**

There are various other more specialized topics related to mining data series, such as segmentation and prediction (forecasting). The interested reader could look at the following surveys [136, 58].

## 2.5 Adaptive Indexing for Interactive Data Exploration

As we've seen so far, most data mining procedures require expensive similarity search computations in order to either identify clusters, anomalies, patterns or to assign labels to sequences. In order to speed up such queries, data series indexes can be used. However, such index structures require expensive initialization time, which could even require days to be completed [30]. It has been recently shown that adaptive indexing can speed up data loading costs and thus improve interactivity. The concept of adaptive indexing was recently introduced in the context of column-store databases [78, 77, 79, 81, 69, 158, 63, 64]. The intuition is that instead of building database indexes up-front, indexes are built during query processing, adapting to the workload. In particular, the algorithms are focused on how to incrementally sort columns in main-memory column-stores. The query predicates are used as pivots during the index refinement steps. Each index refinement step performed during a single query can be seen as a single step of an incremental quick-sort action. As more queries touch a column, this given column reaches closer to a sorted state. The benefit is that adaptive indexing avoids fully sorting columns up front at a high initialization cost, especially when there is no idle time to do so, or no reliable workload knowledge that this is indeed needed. These ideas have also been extended lately for Hadoop-based environments [144].

# Chapter 3

# Minimizing the Data to Query Gap with Adaptive Data Series Indexing

## 3.1 Introduction

In this work, we study the data to query time bottleneck, and focus on the index creation bottleneck for interactive exploration of very large collections of data series. We propose the first adaptive indexing solution for data series, which minimizes the index creation time, allowing users to query the data soon after its generation, and several times faster compared to state-of-the-art indexing approaches. As more queries are posed, the index is continuously refined and subsequent queries enjoy even better execution times.

During creation time, our Adaptive Data Series index (ADS) performs only a few basic steps, mainly creating the basic skeleton of a tree which contains condensed information on the input data series. Its leaves do not contain any raw data series and remain unmaterialized until relevant queries come. As queries arrive, ADS fetches the relevant data series from the raw data, and moves only those data series inside the index. Future queries may be completely covered by the contents of the index, or alternatively ADS adaptively and incrementally fetches any missing data series directly from the raw data set. When the workload stabilizes, ADS can quickly serve fully contained queries, while as the workload shifts, ADS may temporarily need to perform some extra work to adapt before stabilizing again. In addition, ADS does not require a fixed leaf size; it dynamically and adaptively adjusts the leaf size in hot areas of the index. All leaves start with a reasonably big size to guarantee fast indexing times, but the more a given area is queried, the more the respective leaves are split into smaller

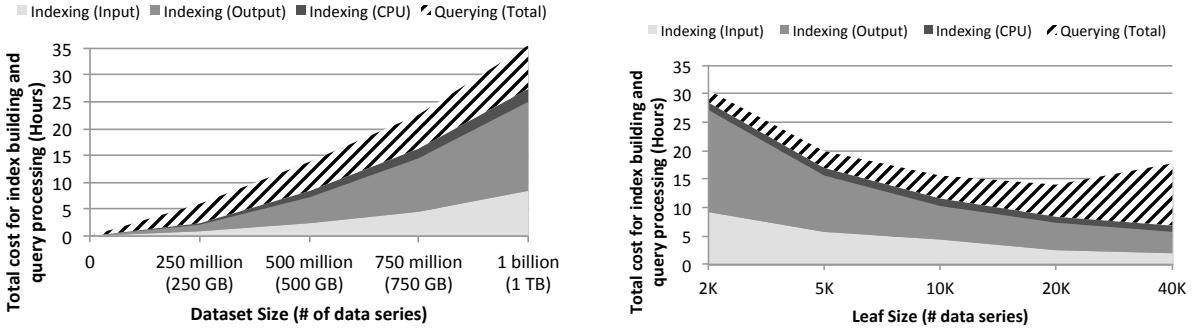
ones to enhance query answering times.

The net effect is that users do not have to wait for extended periods of time before getting access to the data. Our results show that by the time state-of-the art indexing approaches are still in the indexing phase (having answered zero queries), our proposed approach allows users to answer several hundreds of thousands of queries.

Although the concept of adaptive indexing has been studied in the context of column-store databases, there the main goal is to incrementally sort individual arrays (i.e., columns) for point or range queries over 1-dimensional points. In contrast, a data series index is a tree-based index that is tailored to answer similarity search queries over data series collections, thus requiring very different techniques, able to simultaneously index multiple arrays (i.e., data series).

### 3.1.1 Contributions

- We demonstrate the inability of state-of-the-art indexing to cope with exploratory analysis of very large data series collections. We show that the index creation time is a major bottleneck which becomes exponentially worse as data grows.
- We introduce the first adaptive data series index. Adaptive data series indexing minimizes the data to query gap by delaying actions until they are absolutely necessary. Initialization cost is kept at very low levels; only a minimal tree structure based on a summary of the data is built initially. Then, the index structure is continuously enriched as more data and queries arrive and only for the hot part of the data. Each query that is not covered by the current contents of the index, triggers a sequence of actions that have as a side-effect more data to be brought inside the index.
- We demonstrate that no special set-up is required regarding critical low-level details such as leaf size and tree depth. We propose adaptive data series indexing algorithms that start with a rather big leaf size and a shallow tree in order to minimize initialization costs for new data, but then as queries arrive and focus to specific data areas, they adaptively and automatically expand hot subtrees and adjust leaf sizes in the hot branches of the index to minimize querying costs.
- We present algorithms for both approximate and exact query answering. In both cases, we make sure that new data are loaded in the index at a controlled rate (by limiting the number of leaves that are materialized). This is particularly useful as we want to amortize the index creation cost over multiple queries. For the exact




(a) The data to query gap: building a state-of-the-art index and answering $10^5$ queries for big data series collections.

(b) The indexing to querying trade-off: bigger leaf sizes improve indexing speed, but penalize query answering times.

Figure 3.1: The indexing bottleneck.

search, we describe an algorithm that clearly departs from traditional approaches. Existing exact query answering algorithms suffer from a potentially large number of random disk accesses, because of the need to visit leaves on a most-promising-first order. In contrast, the exact algorithm we propose ensures a sequential disk access pattern. It starts by computing lower bounds based on a summarized version of the data (that fits in main-memory), leading to a skip-sequential access pattern on the raw data on disk.

- We experimentally evaluate our approach using both synthetic and real-world datasets, and demonstrate a drastic reduction in the data to query time. The approximate search algorithm is able to handle several hundreds of thousands of queries by the time that state-of-the-art data series (iSAX 2.0 [30]) and multi-dimensional (R-trees [68], X-trees [19], KD-Trees [18]) techniques are still in the index creation phase. Moreover, we show that our approach is faster than the state of the art, also for the task of full index creation.

## 3.2 Minimizing the Data to Query Gap

For big data exploration, it is prohibitive to rely to full sequential scans for every single query, and therefore, indexing is required. The target of indexing techniques is to make query processing efficient enough, such that the analysts can repeatedly fire several exploratory queries with quick response times. However, we show in our work that the amount of time required to build a data series index can be a significant bottleneck;

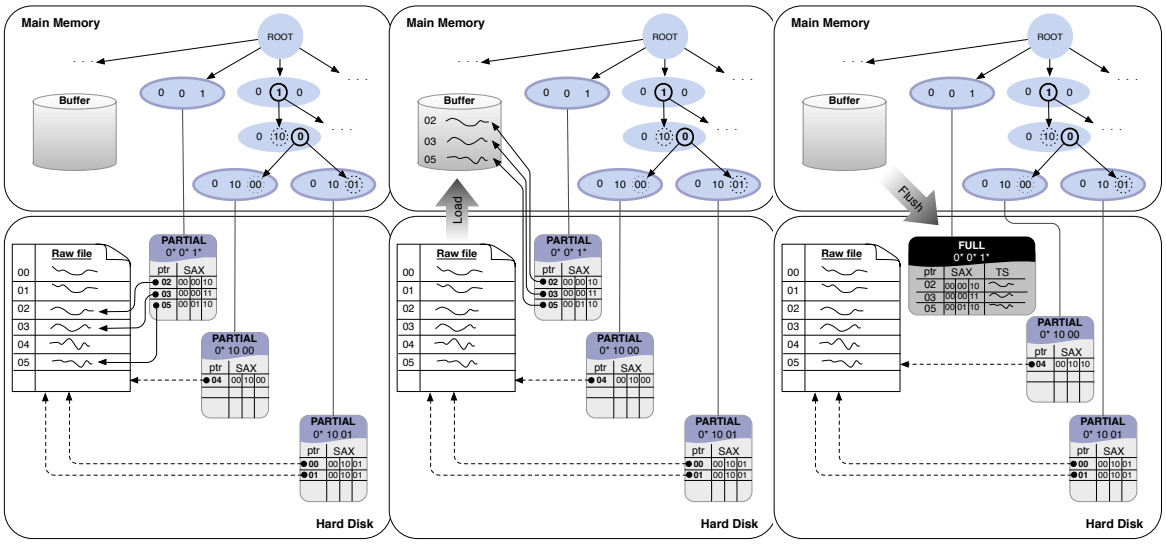
Figure 3.1(a) shows that it takes more than a full day to build a state-of-the-art data series index (iSAX 2.0 [30]) over a data set of 1 billion data series in a modern server machine. The main cost components of indexing are reading the data to be indexed, spilling the indexed data and structures to disk, as well as incurring the computation costs of figuring out where each new data entry belongs to (in the index structure). As the data size grows, the total indexing cost increases dramatically, to a degree where it creates a big and disruptive gap between the time when the data is available and the time when one can actually have access to the data.

As Figure 3.1(b) shows (for a 500 million data series set), the smaller the leaf size is the harder it becomes to build an index, while the bigger the leaf size is, the more we penalize query answering times ($10^5$ queries in this case). Thus, simply choosing a large leaf size does not resolve the data to query problem. To attack this problem, we propose the first adaptive data series index, specifically tailored to solve the problem of indexing and querying very large data series collections. The main idea is that instead of building the complete index over the complete data set up-front and querying only later, we interactively and adaptively build parts of the index, *only for the parts of the data on which the users pose queries*. The net effect is that instead of waiting for extended periods of time for the index creation, users can immediately start exploring the data series. In the next section, we present a detailed design and evaluation of adaptive data series indexing over both synthetic data and real-world workloads. We present three versions of our adaptive data series index: ADS, ADS+ and PADS+, and use a synthetic and 4 real datasets. Our data include DNA sequences, astronomical light-curves, SIFT vectors representing images, and seismic data series. The results show that our approach can gracefully handle large data series collections, while drastically reducing the data to query delay. Indicatively, by the time state-of-the-art indexing techniques finish indexing 1 billion data series (and before answering even a single query), adaptive data series indexing has already answered $3 * 10^5$ approximate queries.

## 3.3 The Adaptive Data Series Family of Indexes

As we discussed earlier, dealing with very large amounts of data series leads to new challenges in data series indexing. Specifically, we stressed the fact that state-of-the-art indexing mechanisms need a prohibitively large amount of time to build a full index: it may take up to several days to create a single index.

In this section, we describe our solution to this problem. We present adaptive data series indexing in detail, and describe how it can reduce the data to query gap by




(a) Initial state. (b) Buffered leaf state. (c) Flushed leaf state.

Figure 3.2: The ADS index states during query answering.

*shifting costly index creation steps from the initialization time to the query processing time*. For ease of presentation, we discuss adaptive data series indexing in two steps; initially we present ADS, a design which introduces the concept of adaptively and incrementally loading data series in the index. Then, we discuss ADS+ which introduces the concept of adaptive splits and adaptive leaf sizes. Finally, we present PADS+, an aggressive variation of ADS+, which is tailored for even better performance in skewed workloads.

### 3.3.1 The Adaptive Data Series (ADS) Index

In order to increase the exploration ability we need to decrease the data to query time. That is, we need to decrease the amount of time needed until a user can access and query new data with good response time. The main bottleneck is the index construction overhead. ADS attacks the index construction bottleneck by shifting the construction of the leaf nodes of the index (the only nodes that can carry raw values for the data series, and have to be stored on disk) to query time. During the index creation phase, ADS creates a tree which contains only the iSAX representation for each data series; the actual data series remain in the raw files and are only loaded in an adaptive way if a relevant query arrives. On the contrary, state-of-the-art indexes, such as iSAX 2.0, a priori load all raw data series in the index at the leaves of the tree (in order to reduce random I/O during query processing). The analysis of the performance of iSAX 2.0 in Figure 3.1(a) motivates our design choice for ADS; it shows that reading from and writing to disk is the main cost component during the indexing phase of iSAX 2.0.

The results show that a big part of these read and write costs is due to reading the raw data series from disk and to writing the leaves of the index tree back to disk (after insertions). Motivated by data exploration scenarios where we do not know a priori which data series are relevant for our analysis, ADS avoids these costs completely at initialization time; it pays such costs at query time, only when absolutely necessary, and only for the data which are relevant to the workload. Below we describe ADS in detail.

**Index Creation**

The index creation phase takes place before queries can be processed but it is kept very lightweight. The process can be seen in Algorithm 22. The input is a raw file which contains all data series in ASCII form. ADS builds a minimal tree during this phase, i.e., a tree which does not contain any data series. The tree contains only iSAX representations. The process starts with a full scan on the raw file to create an iSAX representation for each data series entry. This can be seen in lines 2-5 of Algorithm 22. For data series we also record its offset in the raw data file so future queries can easily retrieve the raw values. To minimize random memory access and random I/O we use a set of buffers in main memory (line 6) to temporarily hold data to be added in the index. When these buffers are full (line 7), we move the data to the appropriate leaf buffer in the index (see discussion in Buffering later on). If necessary, we perform split operations on the way (lines 12-15). The split operation is described in detail in Algorithm 8. Then we sequentially flush each leaf buffer to the disk (Algorithm 22, line 20), set each leaf to be in PARTIAL mode which means that we do not store any raw data series in this leaf (line 21). This process continues until we have indexed all raw data series. We will discuss how we handle new data (updates) later on.

**Delaying Leaf Construction.** The actual data series are only necessary during query time, i.e., in order to give a complete and correct answer. During the index creation time, the iSAX representations are sufficient to build the index tree. In addition, not all data series are needed to answer a particular set of queries. In this way, ADS first creates all necessary iSAX representations and builds the index tree without inserting any data series and only adaptively inserts data series during query processing (to be discussed later on). There are numerous benefits that come with such a design decision, the most important being the significantly reduced cost to build the index. While it is clear that materializing leaves on demand will incur a large random I/O cost, the main benefit comes from the fact that (a) ADS avoids dealing with the raw data series (i.e., other than the single scan on the raw file to create the iSAX representations), (b)

it does not move the raw data series through the tree, and it (c) it does not place the raw data series into the leaf nodes. The data series simply stay in the raw file. This brings benefits in terms of I/O and memory bandwidth used during indexing. Especially when ADS comes to the point of spilling leaf nodes to disk (i.e., all leaves when there is no more free memory), it has a big advantage in that its leaf nodes are very lightweight, containing only iSAX representations, which can be orders of magnitude smaller than the data series themselves. For example, a data series of 256 points with a float precision of 4 bytes, can be efficiently summarized with 16 characters of 1 byte each. Moreover, by not inserting the data series in the index, we significantly reduce the cost of splits at the leaf level during the indexing phase; the I/O cost is minimized as only iSAX representations are shuffled between index nodes. All ADS variations maintain the main index tree in memory, while leaf nodes are kept on disk.

**Buffering.** ADS improves locality when inserting data by buffering data at two levels of the index. Buffering amortizes random access (both in memory and on disk) and is a common practice to improve locality in tree-based indexes, e.g., [193, 30], or even in database query plans (which typically have a tree shape) [194]. During index creation, instead of pushing iSAX representations through the index one at a time, ADS initially keeps those in the First Buffer Layer (FBL), a set of buffers corresponding to the children nodes of the index root. Once the FBL is full (i.e., all free memory is consumed), these representations are then passed through the tree and moved to the second layer of buffers corresponding to the leaf nodes of the index, called Leaf Buffer Layer (LBL). Data is then flushed to disk one leaf at a time, ensuring sequential writes. Additionally, every time that a leaf needs to be split and iSAX representations need to be read from disk, we keep them in the LBL, until we run out of space (Algorithm 8, lines 1-2). The leaves are flushed again when there is no more free memory.

**Mapping on the Raw File.** ADS reduces the index creation costs by not keeping around the data series. However, the raw data series is needed when queries arrive. For this reason, ADS needs an efficient way to quickly access a given data series entry. To achieve this, ADS maintains a single pointer for each data series entry $X$ in the leaf node where data series $X$ would normally reside. This is a pointer to the raw data file that provides direct access to the raw data series. (As we will discuss later on, the first time the leaf is accessed by a query all pointers are dropped and the corresponding raw data series are loaded.)

**Example.** An example of ADS is shown in Figure 3.2; the figure depicts the state of the index after certain events. An index is built on top of a set of iSAX words with a word size of 3 characters and a maximum cardinality for each character of 2 bits. The

**Algorithm 1:** createIndex(**file**, **index**, **n**)

```
1  while not reached end of file do
2      position = current file position;
3      dataSeries = read data series of size n from file;
4      isax = convert dataSeries to iSAX;
5      Move file pointer n points;
6      Add the (isax, position) pair in the index's FBL buffer;
7      if the main memory is full then
8          // Move data from the First Buffers (FBL)
9          // to the appropriate Leaf Buffer (LBL)
10         for every (isax, position) pair ∈ FBL buffer do
11             targetLeaf = Leaf of index for putting (isax, position);
12             while targetLeaf is full do
13                 Split(targetLeaf, isax);
14                 targetLeaf = New leaf for putting (isax, position);
15             Insert (isax, position) in targetLeaf's LBL buffer;
16         // Flush all Leaf Buffers containing
17         // (isax, position) pairs to the disk, and
18         // set them in PARTIAL mode (no raw data)
19         for every leaf in index do
20             Flush the LBL buffer of this leaf to the disk;
21             Set leaf to be in PARTIAL mode;
22         clear buffers;
```

leaf nodes are depicted as oval shapes with border lines and the intermediate nodes without any border lines. Each intermediate node is split on a single character; the one surrounded by a bold cycle. Each leaf node is connected to a file on disk, where the full cardinality iSAX representations and the corresponding pointers to the raw file are stored. Figure 3.2(a) shows how the index looks like immediately after the initialization phase and before any query has been processed. In this case, all leaf nodes are in PARTIAL mode, i.e., they do not contain any data series, since no query has been executed yet. Figure 3.2(b) and Figure 3.2(c) show what happens when a query arrives and we discuss that in the next subsection.

**Algorithm 2:** Split(**leaf**)

```
1 diskData = get data from leaf's disk pages;
2 Insert diskData in leaf's buffer (LBL buffer);
3 Split leaf in the best point and create two new children leaves;
4 Set leaf as an intermediate node;
5 Set leaf.leftChild in PARTIAL mode;
6 Set leaf.right in PARTIAL mode;
7 for every (isax, position) pair ∈ leaf's LBL buffer do
8 |   Insert (isax, position) pair in the appropriate child leaf;
```

**Algorithm 3:** approxSearchADS(**dataSeries**, **isax**, **index**)

```
1 targetLeaf = leaf of index where this isax should be inserted;
2 // Calculate the real leaf distance between the dataSeries
3 // and the raw data series that this leaf refers to or contains.
4 bsf = calculateRealLeafDistance(targetLeaf, dataSeries);
5 return bsf;
```

#### Querying and Refining ADS

We continue our discussion by describing the process of query answering using ADS. Contrary to static indexes, the querying process in ADS contains a few extra steps. In addition to answering a query $q$, the query process refines the index during the processing steps of $q$. These extra index refinement steps do not take place after the query is answered; they develop completely on-the-fly and are necessary in order to answer $q$. At any given time, ADS contains just enough information in order to handle the current workload. Thus, when new queries arrive, which do not follow the patterns in previous requests, ADS needs to enrich the index with more information.

We provide algorithms for both approximate search and exact search. Approximate search provides answers of good quality (returns a top 100 answer for the nearest neighbor search in 91.5% of the cases for iSAX [165, 166]) with very fast response times. On the other hand, exact search guarantees that we get the exact answer, but with potentially much higher query execution time.

**Approximate Search.** When a query arrives (in the form of a data series), it is first converted to an iSAX representation. Then, the index tree is traversed searching for a leaf with an iSAX representation similar to that of the query (Algorithm 5). This is

**Algorithm 4:** exactSearchADS(**dataSeries**, **index**)

```
1  isax = convert dataSeries to iSAX;
2  bsf = approxSearchADS(dataSeries, isax, index);
3  bsfDist = Infinite;
4  queue = Initialize a priority queue with the root nodes of the index;
5  while node = pop next node from queue do
6      if node is a leaf and MinDist(dataSeries, node) < bsfDist then
7          realDist = calculateRealLeafDistance(dataSeries, node);
8          if realDist < bsfDist then
9              bsf = node;
10             bsfDist = realDist;
11     else if MinDist(dataSeries, node) ≥ bsfDist then
12         // Found the nearest neighbor, break the loop
13         break;
14     else
15         // It is an intermediate node: push children to the queue.
16         minDLeft = MinDist(dataSeries, node.leftChild);
17         minDRight = MinDist(dataSeries, node.rightChild);
18         if minDLeft < bsfDist then
19             Put node.leftChild in queue with priority minDLeft;
20         if minDRight < bsfDist then
21             Put node.rightChild in queue with priority minDRight;
22 return bsf;
```

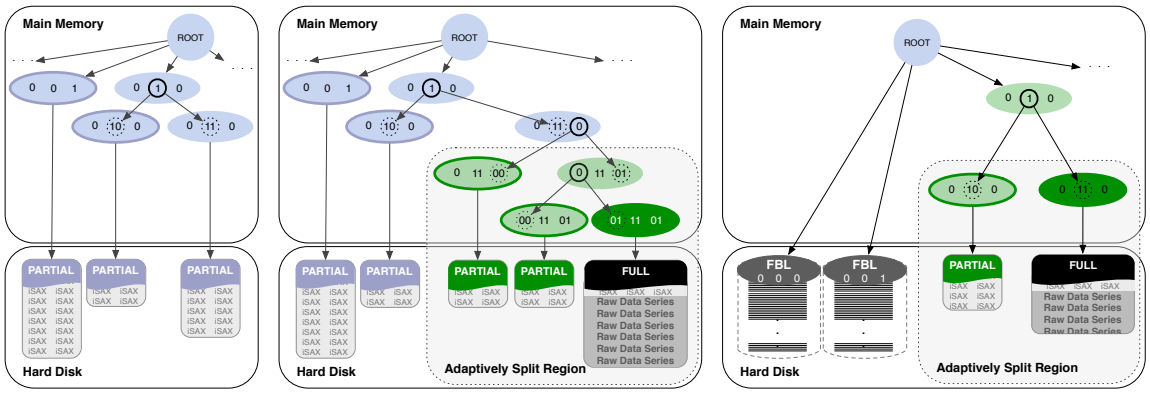
the leaf where the query series would reside if it was a part of the indexed dataset. Whether such a leaf exists already or not, depends not only on the data, but also on past queries. If such a leaf does not exist, then the most similar leaf to the query is used instead. In the case that the leaf node where the search ends is in PARTIAL mode, i.e., it contains only iSAX representations but not any data series, then all missing data series are fetched from the raw file. To enrich a partial leaf, ADS fetches the partial leaf from disk and reads all the positions in the raw file of the data series that belong in this leaf. (A partial leaf holds the iSAX representation for each data series and also its position in the raw file.) Then, it sorts those positions (to ensure sequential access to the raw file) and fetches the raw data series. The new data series are assigned to leaf nodes and kept in memory in the LBL buffers (Figure 3.2(b)). The corresponding leaf node

**Algorithm 5:** calculateRealLeafDistance(**leaf**, **dataSeries**)

```
1  // Check if the raw data have been fetched in the leaf
2  if leaf is in FULL mode then
3      if leaf has raw data series in LBL buffer then
4          bufferBSF = find closest to dataSeries record in LBL;
5      if leaf has raw data series on disk then
6          diskBSF = find closest to dataSeries record on disk;
7      if diskBSF < bufferBSF then
8          return diskBSF;
9      else
10         return bufferBSF;
11 else if leaf is in PARTIAL mode then
12     // Materialize leaf
13     records = Get all (isax, position) pairs from disk and LBL;
14     Sort records based on positions;
15     for every (isax, position) pair ∈ records do
16         Seek position in raw data file;
17         rawDataSeries = Fetch raw data series from raw data file;
18         Insert (isax, position, rawDataSeries) tuple in LBL buffer;
19         if main memory is full then
20             Flush all LBL buffers on disk;
21     Set leaf to FULL mode;
22     return calculateRealLeafDistance(node, dataSeries);
```

contains pointers to the buffered data. When there is no more free memory, the LBL buffers are flushed to disk (as seen in Figure 3.2(c)). The corresponding leaf is then marked as FULL. At this point the leaf data is fully materialized and future queries that need to access the data series for this leaf node, need to fetch the binary leaf data from disk or from the LBL buffer. Once the data series that match the current query are available (either being fetched from the raw file, from the buffer, or from disk) then the real distance from the query is calculated. The minimum distance found in the leaf is used as the approximate answer.

**Exact Search.** When a query arrives, an approximate search is initially issued in order to get an initial Best So Far answer (BSF). If the BSF is not 0, which means that we did not find a perfect match, then the node with the best possible answer has to




(a) ADS+ after indexing. (b) ADS+ index after a query. (c) PADS+ index after a query.

Figure 3.3: Examples of ADS+ and PADS+ states.

be identified. This is done in a recursive way as in the original iSAX index using the MinDistPaaToiSAX [165], and until we are not able to improve BSF any further. The difference is that if a new leaf is needed, which is in partial mode, ADS will enrich this leaf on-the-fly.

The algorithm, described in Algorithm 4, starts by putting all the children of the root in a min-stack ranked using their lower distance bound towards the query (line 4). Then the one with the best minimum distance is popped (line 5) and explored, as long as this distance is better than BSF (lines 6-10). If the currently popped node is an intermediate node (lines 14-21) then its children are pushed into the min-stack for possible future exploration. The process continuous recursively, and stops when the best lower bound is bigger than the BSF distance (lines 11-13), which means that it is not possible to improve the current answer any further.

**Example.** Continuing the example of Figure 3.2, Figure 3.2(b) and Figure 3.2(c) show what happens when a query arrives. Figure 3.2(b) depicts the case when a query reaches a non materialized leaf. The raw data series are fetched in main memory buffers, and the leaf now points to them. If the buffers become full, the raw data series for each leaf are flushed to disk, thus converting them into fully materialized leaves. This can be seen in Figure 3.2(c); the full leaf contains both the iSAX representations and the raw data series.

### 3.3.2 The ADS+ Index (Adaptive Leaf Size)

ADS drastically reduces the index creation time by avoiding the insertion of raw data series in the index until a relevant query arrives. However, there is opportunity for significant further optimizations; by studying the operations that get executed during adaptive index building and refinement we found that the time spent during split operations in the index tree is a major cost component.

**Leaf Size and Splits.** Splits are expensive as they cause data transfer to and from disk (to update node data). The main parameter that affects split costs is the leaf size, i.e., a tree with a big leaf size has a smaller number of nodes overall, causing less splits. Thus, a big leaf size reduces index creation time. However, as we have shown in Figure 3.1(b), big leaves also penalize query costs and vice versa: when reaching a big leaf during a search, we have to scan more data series than with a small leaf. State-of-the-art indexes rely on a fixed leaf size which needs to be set up front, during index creation time, and typically represents a compromise between index creation cost and query cost.

**Adaptive Leaf Size.** To further optimize the data to query time, we introduce a lightweight variation of ADS, *ADS+*, with a more transparent initialization step. The main intuition is that one can quickly build the index tree using a large leaf size, saving time from very expensive split operations, and rely on queries that are then going to force splits in order to reduce the leaf sizes in the hot areas of the index. ADS+ uses two different leaf sizes: a big build-time leaf size for optimal index construction, and a small query-time leaf size for optimal access costs. This allows us to make future queries benefit from every split operation performed, finding the relevant data by traversing the tree, and not by scanning larger leaves. Initially, the index tree is built as in plain ADS (Algorithm 22), with a constant leaf size, equal to build-time leaf size. In traditional indexes, this leaf size remains the same across the life-time of the index. In our case, when a query that needs to search a partial leaf arrives, ADS+ refines its index structure on-the-fly by recursively splitting the target leaf, until the target sub-leaf becomes smaller or equal to the query-time leaf size. This can be seen in Algorithm 5. Additionally both Approximate and Exact search have been modified to use this policy, a shown in Algorithm 9 (lines 2-5) and Algorithm 28 (lines 7-10), respectively.

Intuitively what happens is that the target leaf is split until it becomes small enough, while all leaves created due to split actions but are not needed for this query are then left untouched and thus with a leaf size which is between the big construction-time leaf size and the small query-time leaf size. If and only if the workload shifts and future

**Algorithm 6:** SplitADS+(**leaf**, **targetLeafSize**)

```
1 /* If the leaf size is bigger than the target leaf size, split node. */
2 if leaf's leaf size > targetLeafSize then
3     Split(node);
4     SplitADS+(node.leftChild, targetLeafSize);
5     SplitADS+(node.rightChild, targetLeafSize);
```

**Algorithm 7:** approxSearchADS+(**dataSeries**, **isax**, **index**, **queryTimeLeafSize**)

```
1 targetLeaf = leaf of index where this isax should be inserted;
2 if targetLeaf's leaf size > queryTimeLeafSize then
3     // It can be additionally split
4     SplitADS+(targetLeaf, queryTimeLeafSize);
5     targetLeaf = targetLeaf's descendant where this isax should be inserted;
6 // Calculate the real distance between the dataSeries
7 // and the raw data series that this leaf points to.
8 bsf = calculateRealLeafDistance(targetLeaf, dataSeries);
9 return bsf;
```

queries need to query those leaves, then ADS+ automatically splits those leaves even further to reach a leaf size that gives good query processing times.

**Example.** An example of this process is shown in Figures 3.3(a) and 3.3(b). Figure 3.3(a) depicts the state of ADS+ after initialization and before any query has arrived, while Figure 3.3(b) shows how a single query results in adaptive splits of the right sub-tree until the target leaf node is fully materialized; intermediate nodes remain in partial mode and with a variable leaf size.

Adaptive and on demand leaf splitting allow ADS+ to have both fast index building and fast query processing. It does not waste time on creating fine-grained versions of each sub-tree of the index, but rather concentrates on the parts that are related to the current workload. When queries focus to a subset of the dataset, ADS+ does not need to exhaustively index and optimize all data; it rather concentrates on the most related sub-trees of the index. When the workload shifts and a new area of the index becomes relevant, then the first few queries adaptively optimize the index for the new area as well by expanding the proper sub-trees and adjusting leaf sizes.

**Delaying Leaf Materialization.** Another optimization that gives ADS+ a lightweight behavior is that it delays leaf materialization even further. In particular, when travers-

**Algorithm 8:** exactSearchADS+(**dataSeries**, **index**, **queryTimeLeafSize**)

```
1  isax = convert dataSeries to iSAX;
2  bsf = approxSearchADS+(dataSeries, isax, index, queryTimeLeafSize);
3  bsfDist = Infinite;
4  queue = Initialize a priority queue with the root nodes of the index;
5  while node = pop next node from queue do
6      if node is a leaf and MinDist(dataSeries, node) < bsfDist then
7          if node's leaf size > queryTimeLeafSize then
8              // Need to split this leaf more
9              SplitADS+(node, queryTimeLeafSize);
10             Re-Insert node in queue;
11         else
12             // No need to split any more
13             dist = calculateRealLeafDistance(dataSeries, node);
14             if dist < bsfDist then
15                 bsf = node;
16                 bsfDist = dist;
17     else if MinDist(dataSeries, node) ≥ bsfDist then
18         // Found the nearest neighbor, break the loop
19         break;
20     else
21         // It is an intermediate node: push children to the queue.
22         minDLeft = MinDist+(dataSeries, node.leftChild);
23         minDRight = MinDist+(dataSeries, node.rightChild);
24         if minDLeft < bsfDist then
25             Put node.leftChild in queue with priority minDLeft;
26         if minDRight < bsfDist then
27             Put node.rightChild in queue with priority minDRight;
28 return bsf;
```

ing the tree for query processing, which leads to adaptive leaf splitting, ADS+ does not materialize the initial big leaf, nor all the leaves it creates on its way to the target small leaf. For example, when ADS+ needs to split big leaf $X$ and this results in $X$ being split recursively into $n$ new nodes until we reach the target leaf $Z$ with a small leaf size, ADS+ fully materializes only leaf $Z$. For the rest of the leaves it uses the partial

**Algorithm 9:** MinDist+(**dataSeries**, **leaf**)

```
1  if leaf is in FULL mode then
2      /* Use the coarse SAX representation of all the data series and calculate the minimum
       distance .*/
3      return MinDist(dataSeries, leaf);
4  else
5      /* The node is not materialized yet. We can load the small iSAX representations file and
       calculate a tighter minimum distance using the iSAX representations of all the data series. */
6      isaxValues = Get all isax representations from disk and LBL;
7      maxMinDist = 0;
8      for isax ∈ isaxValues do
9          minDist = MinDist(dataSeries, isax);
10         if minDist > maxMinDist then
11             maxMinDist = minDist;
12     return maxMinDist;
```

information contained in the leaves to perform the splits, i.e., the iSAX representations. This results in (a) less computation as opposed to having to split based on raw data, (b) less I/O as SAX representations are much smaller, and (c) it enhances the adaptive behavior of ADS+ as it materializes only the truly interesting data that the queries are targeting.

### 3.3.3 Partial ADS+ (PADS+)

Although the ADS variations described above help to reduce the indexing cost by omitting the raw data from the index creation process, ADS and ADS+ still need to spend time for creating the basic index structure. This means that users still have to wait until this process finishes, and even though it is a much faster process than full indexing, still certain applications may want *even faster* access to their data. To further optimize the data to query time, we introduce a more lightweight technique which extends ADS+ with an even more transparent initialization step. It is tailored for scenarios where users may want to fire just a few approximate queries, as well as for scenarios with high query workload skew. The new approach is named *Partial ADS+ (PADS+)* and its main intuition is to gradually build parts of the index tree, and only for small subsets of the data as queries arrive. The concept is similar to the idea of partial indexes [170] with the difference that the index is not static, i.e., it is not defined for a pre-decided

set of the data; instead it continuously evolves to fit the workload.

**Index Initialization.** The initialization step of PADS+ is kept as lightweight as possible. PADS+ does not build an index tree at all; there is only a root node with a set of FBL buffers that contain only the iSAX representations. The only step which takes place during the initialization phase is that PADS+ creates the iSAX representations based on the raw data (as in ADS+). This requires a complete scan of the raw data. But then, instead of spending a significant effort using the SAX representations to create a tree as ADS+ does, PADS+ stops at this point and is ready to process queries. The iSAX representations are first kept in in-memory FBL buffers and then (contrary to ADS and ADS+) spilled to disk when the buffers are full. Since these buffers persist on disk, we refer to them as *FBL persistent-buffers (FBL p-buffers)*. All these steps are similar to a subset of the initialization effort that takes place for ADS+. This approach allows PADS+ to significantly reduce the data-to-query time.

**Adapting to Queries.** PADS+ continuously and incrementally is refined as queries arrive. As the workload shifts and requires new data areas, the nodes in the index tree are adaptively and recursively split to smaller nodes that contain the required data. It follows the same procedure as with ADS+ with the difference that the starting point is an index with a just single root node with no children nodes. In this way, only the parts of the index which are truly relevant for the workload are further developed as queries arrive.

**Skewed Workloads.** Such an adaptive design favors scenarios where there is high skew in the workload, i.e., only part of the dataset is interesting, or when there is periodical skew in the sense that queries focus on a single area of the domain for a given time before the focus shifts to another area.

**Querying.** When a query is issued, PADS+ converts the query to its iSAX representation and finds the corresponding FBL p-buffer. It then loads the iSAX representations and adaptively splits the buffer data, until the query-time leaf size is reached, at which point it loads the raw time series for that leaf. This process is repeated during query answering, performing adaptive splits every time that algorithm has to calculate the distance to a leaf node that has not yet been split to the query-time leaf size.

When the query answering algorithm needs data that are missing from the tree, it needs to scan the data of the corresponding FBL p-buffer and perform an adaptive split operation on it. In this process, the initial leaf size is set to infinite; thus, adaptive split operations can be performed by splitting the large buffers and creating large leaf files, which are split again *only* if there is a query that asks for them.

Furthermore, using 16 PAA segments (which is common in practice), we initially

have $2^{16}$ FBL buffers. As a result, given a dataset of 1 billion data series, each one of the 65536 FBL buffers will on average contain around 15 thousand iSAX representations. Using a 1 byte representation for each iSAX character (i.e., cardinality 256), and given the fact that we have 16 segments, we would need 16 bytes for representing each data series. This means that the average FBL size would be around 235 KB: a file size that makes it trivial to perform split operations on.

**Example.** An illustration of the PADS+ index can be seen in Figure 3.3(c). It represents a random instance after a few queries have arrived. The index is not fully built; only a small part of the index is created and only some of the leaves are materialized, following the workload. For example, the two leftmost children of the root point directly to FBL p-buffers on disk; no query has gone through this path. On the contrary, the rightmost child of the root is split, leading to a subtree which reaches down to two leaf nodes. This subtree is created as a side-effect of a query requesting for data series that belong in the leaf that is now marked as FULL in Figure 3.3(c).

### 3.3.4 Handling Updates in ADS, ADS+ and PADS+

Efficiently supporting index updates is an important problem that has gathered a lot of attention [61, 9]. ADS has been designed to efficiently support updates (insertions/deletions), as well. This process can be seen in Figure 3.4, where we depict the status of the index after a series of operations, involving insertions, leaf materializations and deletions.

**Inserts.** Insertions is the main scenario of interest in data exploration environments, i.e., in a scientific database new data is continuously created, but past data is not discarded. Handling inserts in all ADS variations is done by simply appending the new data series in the raw file, while only its iSAX representation and its position in the raw file is pushed through the index tree. If the index leaf has already been materialized and is in FULL mode, we create an additional PARTIAL file where the new data series reside. We then set a bit that informs us that the PARTIAL file is not empty. No further actions are needed for partial leaves. If a future query reaches a FULL leaf with pending inserts, then it fetches the new inserts on-the-fly and merges them in the leaf in the same way it is done for PARTIAL leaves (as we discussed earlier).

Figure 3.4 illustrates the example of a single leaf in the presence of several updates. Initially this leaf is empty, and at time $t1$, data series 01, 02 and 03 are inserted. During $t2$ the leaf is materialized (as a result of a query that had to access it). This results in the above three data series to be part of the FULL file after $t2$. In $t3$, two more data series are inserted, and as a result they end up in the PARTIAL file. Since no query has

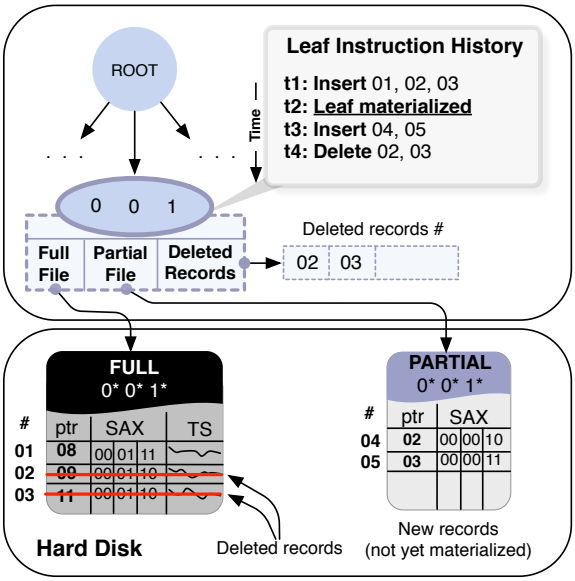



Figure 3.4: Insertions and Deletions in ADS.

accessed this leaf after $t3$, the new data series is not materialized.

**Deletes.** When a data series needs to be deleted, we simply mark the data series as deleted in its corresponding leaf (via an in-memory per-leaf bit-vector). Whether the leaf is partial or full does not make a difference. Future queries ignore deleted data series, while future insertions can exploit the space created in this leaf by these ghost entries.

This case is also depicted in the example of Figure 3.4, where at time $t4$, data series 02 and 03 are deleted. We update the in-memory bit-vector of deleted records to include these two data series. Therefore, the data series are marked as deleted and their locations can be overwritten with new data.

In the case where a leaf becomes completely empty, we destroy the leaf and clear the memory that it occupies. If that leaf had no siblings, the parent node is also deleted. This process is propagated upwards until we reach a node that has a non-empty sibling.

## 3.4 Experimental Evaluation

In this section, we present our experimental evaluation. We demonstrate that adaptive data series indexing drastically reduces the initialization time, achieving up to one order of magnitude smaller data-to-query time when compared to state-of-the-art ap-

proaches. We show that our algorithms enable users to perform hundreds of thousands of queries faster, while the index creation cost is spread across multiple queries.

**Algorithms.** We benchmark all indexing methods presented in this paper and we compare all our adaptive indexing variations against the state-of-the-art iSAX 2.0 index [30] that supports bulk loading. We also compare against sequential scan, and two state-of-the-art multi-dimensional indexes: R-Trees [68] and X-Trees [19]. Finally, we implemented several main-memory performance optimizations in the iSAX 2.0 code: we use an LRU buffer for recently queried nodes and also after loading we maintain its last loading buffer in memory.

**Infrastructure and Implementation.** All the data structures and algorithms presented, as well as an optimized version of iSAX 2.0, are built from scratch in C and compiled with GCC 4.6.3 under Ubuntu Linux 12.04.2. We used an Intel Xeon machine with 64GB of RAM and 4x 2TB, SATA, 7.2K RPM Hard Drives in RAID0. All algorithms are tuned to make maximum use of all available memory.

**Datasets.** We use several synthetic datasets for a fine grained analysis, as well as 4 real datasets coming from different domains, in order to demonstrate the usefulness of adaptive data series indexing in real-life scenarios.

For the synthetic datasets, we used a random walk data series generator. This is a generator, where a random number is drawn from a Gaussian distribution $N(0,1)$, then at each time point a new number is drawn from this distribution and added to the value of the last number. This kind of data generation has been extensively used in the past [11, 59, 131, 13, 165, 166, 30], and has been shown to effectively model real-world financial data [59].

The real datasets are the following. The first dataset (TEXMEX) [84] contains 1 Billion vectors representing images. The second dataset (DNA) contains 20 Million DNA sequences coming from the Homo Sapiens and Rhesus Macaque genomes [31]. The third dataset (SEISMIC) contains 100 Million seismic data series collected from the IRIS Seismic Data Access repository [8]. Finally, the fourth dataset (ASTRO) contains 200 Million astronomical data series representing celestial objects [168]. Each dataset is $z$-normalized before being indexed. Unless mentioned otherwise, each data series consists of 256 points and each point has a float precision of 4 bytes.

**Workloads.** The query workloads for every scenario are random. Each query is given in the form of a data series $q$ and the index is trying to locate if this data series or a similar one exist in the database. We study query intensive workloads with various patterns, including skewed workloads, as well as update workloads (the details are provided in the description of the experiments).

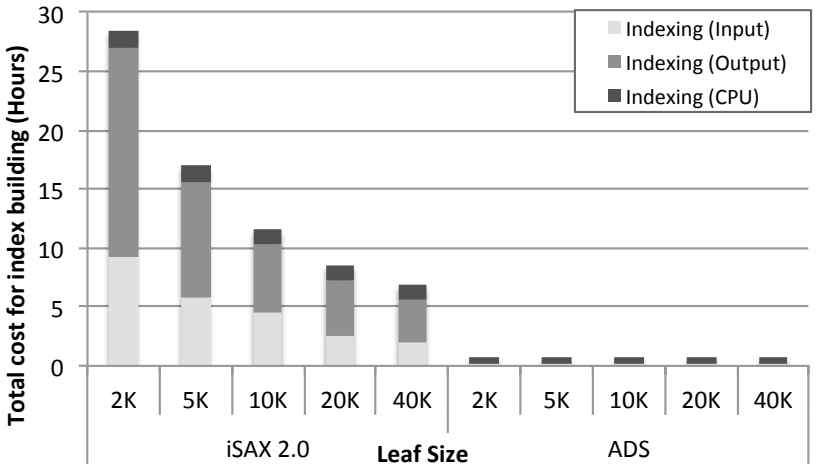



Figure 3.5: Reducing indexing costs.

### 3.4.1 Reducing the Data to Query Time

**Motivation.** In our motivation discussion in the introduction section of the paper, we discussed Figure 3.1(a) as an example that demonstrates the limits of state-of-the-art indexing techniques. For this experiment, we used a synthetic data set of up to 1 billion data series (1 TB) and $10^5$ random queries (73% of which need to fetch new data from the raw file). The main observation is that as we try to index more and more data, the initialization time to build a state-of-the-art data series index becomes a prohibitive factor. With 1 billion data series it takes more than a full day in order to index all data using the state-of-the-art iSAX 2.0 index even when a preferable leaf size is used (Figure 3.1(a)).

**Minimizing Indexing Costs.** Let us now see how the adaptive data series indexing ideas can help in reducing the index building costs. In this experiment, we use the same set-up as before, but we now use a constant data size of 500 million data series and we vary the leaf size. We test iSAX 2.0 against ADS.

Figure 3.5 depicts the results, where we show the total time needed to index all data. ADS drastically reduces the index build time compared to iSAX 2.0 regardless of the leaf size. For example, for the case of a leaf size of 20K data series, which is the best case for iSAX 2.0 (we elaborate on this choice in the following paragraphs), ADS builds the index in only half an hour, while iSAX 2.0 needs 8 hours.

The breakdown of the indexing costs in Figure 3.5 explains this behavior. *Input* is the time spent reading data from disk. *Output* is the time spent writing data to disk. *CPU* is the time spent doing any kind of computation during indexing. ADS avoids

the expensive steps of placing each data series in its corresponding leaf node. The net result is that the Input and Output costs, i.e., the I/O costs, drop drastically compared to iSAX 2.0. At the same time, also the CPU cost drops as ADS does not have to go through the index to place each data series. Overall, reducing the I/O and CPU costs results in a major benefit for ADS during the indexing phase.

**The Query Processing Bottleneck of Plain ADS.** Having seen that ADS can reduce the indexing costs, let us now see the effect on query processing. Figure 3.6 shows the results. Using the same set-up as in the previous experiment, it depicts the total time to build the index and to process all $10^5$ queries. There are two observations from the behavior seen in Figure 3.6. First, ADS allows its first few queries to access the data faster than iSAX 2.0. For example, if we take the best leaf size case for ADS (2K) and the best leaf size case for iSAX 2.0 (20K), we see (marked with the red arrow) that ADS can answer 12,700 queries by the time iSAX 2.0 is still indexing and has not answered a single query (9 hours). In this way, ADS provides a quick gateway to the data as it was the original intention and motivation. However, as we process more and more queries and regardless of the leaf size, ADS looses its initial advantage; queries take too long to process and overall ADS does not present a feasible solution.

The main reason why ADS suffers is that even a single query might result in fetching a significant amount of raw data series. For example, if a query reaches a leaf which is not yet materialized and the leaf size is set to 2K, then ADS needs to fetch 2K raw data series in order to materialize the leaf. Such costs, significantly penalize queries and in the case of random workloads, as in the example of Figure 3.6, where each query may hit a completely different area of the index, this brings a significant overall cost. In a more focused workload, i.e., where queries focus on a given part of the index, the overall performance is drastically different as we do not reach the point where we need to fetch extra raw data very often. We discuss such examples later on.

Still though, ADS does not represent a robust solution, i.e., a solution that would be globally applicable in arbitrary workloads.

**Robustness with ADS+.** This is exactly the motivation for ADS+. ADS+ maintains the adaptive properties of ADS but it is also robust and scalable. To demonstrate this behavior, we repeat the previous experiment, this time using also ADS+. Figure 3.7 shows that ADS+ significantly outperforms iSAX 2.0 not only during the index building phase but also during the query processing phase. For example, for the best case of iSAX 2.0, i.e., with leaf size 20K, ADS+ can create the index and process all $10^5$ queries in only 3 hours while iSAX 2.0 needs roughly 15 hours. In fact, ADS+ can process the queries even faster as it may use even smaller leaf sizes.

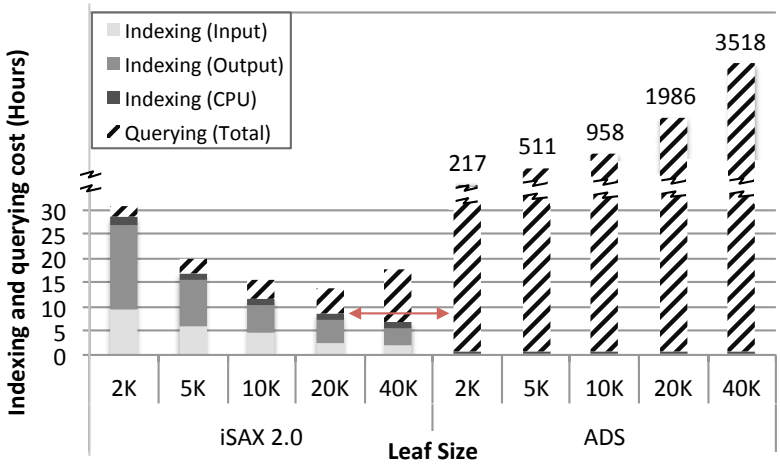



Figure 3.6: The query processing bottleneck.

Next, we show that ADS+ is robust even when in inferior set-up. Using the same set-up (data and queries) as before, we vary the available memory the algorithms can exploit. In addition, for iSAX 2.0 we use a buffer pool with an LRU policy so that it can hold recently visited nodes in memory. In Figure 3.8, it can be seen that even if we use 10% of the main memory for ADS+, it can still answer all of the $10^5$ queries before iSAX 2.0 has finished indexing using 100% of the main memory.

The main novelty in ADS+ is that it can maintain a lightweight index-building step due to only partially building the index but also due to using a large leaf size during this phase. Then, as queries arrive, it adaptively splits leaves in hot areas of the index such that queries in this area may be processed at a smaller cost. In this way, ADS+ solves the robustness and scalability problem of ADS by introducing adaptive node splits, i.e., by being able to adjust the shape of the index based on the workload and only for the areas which are hot and may cause expensive steps for individual queries.

**Choosing the Query-Time Leaf Size.** The query-time leaf size indicates the finest granularity in which we will split a node with ADS+, and consequently it is directly related to the amount of raw data that we store on disk under each leaf. We have experimented with various query-time leaf sizes ranging from 1 data-series to 1000 data-series, and measured the average page utilization for 3 different page sizes, as well as the average query answering time. We did this by running $10^5$ queries on a dataset of 500 million data-series. As we can see in Table 3.1, the smaller the query-time leaf size is, the less data we have to fetch from the raw data file, and the faster the materialization of the leaf node is. On the other hand, very small values of query-

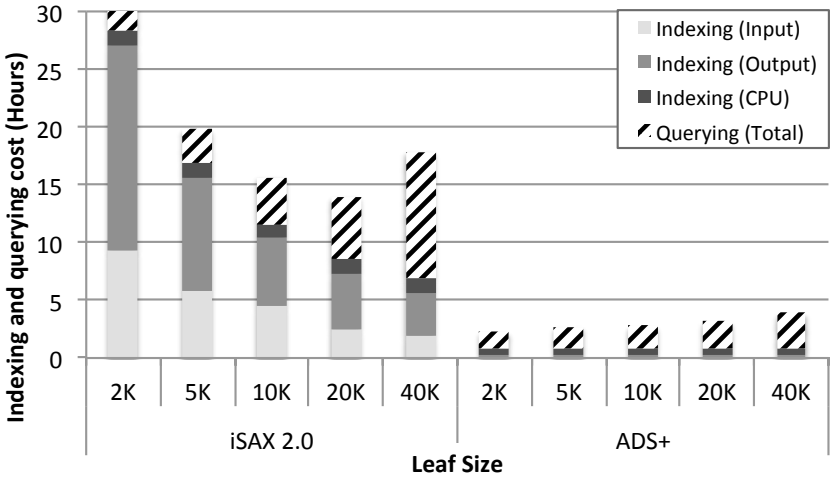



Figure 3.7: Reducing the data-to-query time with ADS+.

time leaf size adversely affect space utilization, since page occupancy will be small. As a result, it is important to choose a leaf size that will allow for the maximum page utilization while at the same time offers an acceptable query answering time. For the rest of our experiments we use 10, since when using a page size of 8KB, we maximize page occupancy at around 89% (Table 3.1 in bold) and the average query answering time remains relatively low at 69 milliseconds.

| **Query-time leaf size** | **1** | **10** | **100** | **1000** |
|---|---|---|---|---|
| **Query time (millisec.)** | 11.27 | 67.64 | 499.95 | 4031.68 |
| **# of 4KB pages** | 0.25 | 1.79 | 17.23 | 171.48 |
| **# of 8KB pages** | 0.12 | **0.89** | 8.61 | 85.74 |
| **# of 16KB pages** | 0.06 | 0.45 | 4.30 | 42.87 |

Table 3.1: Varying query-time leaf size.

**Scaling to 1 Billion Data Series.** Next, we stress all indexing strategies to study how they can cope with an increasing data set size. We study the behavior up to 1 billion data series and with $10^5$ random queries. Regarding leaf sizes, we use the optimal leaf size observed for each index strategy, i.e., 20K for iSAX 2.0, 2K for ADS, and for ADS+ 2K build-time and 10 query-time leaf size. Figure 3.9 shows the total time to build the index and answer all queries. Across all data sizes, ADS+ consistently outperforms all other strategies by a big margin.

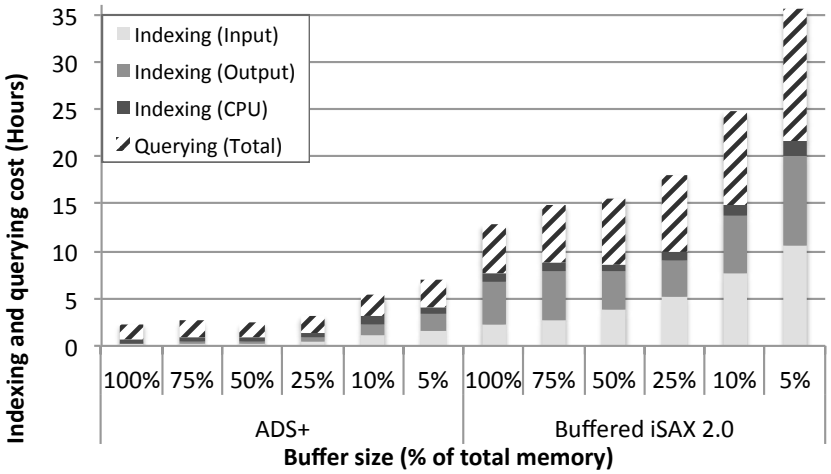



Figure 3.8: Total indexing and query answering cost as we increase the buffer size for ADS+ and iSAX 2.0.

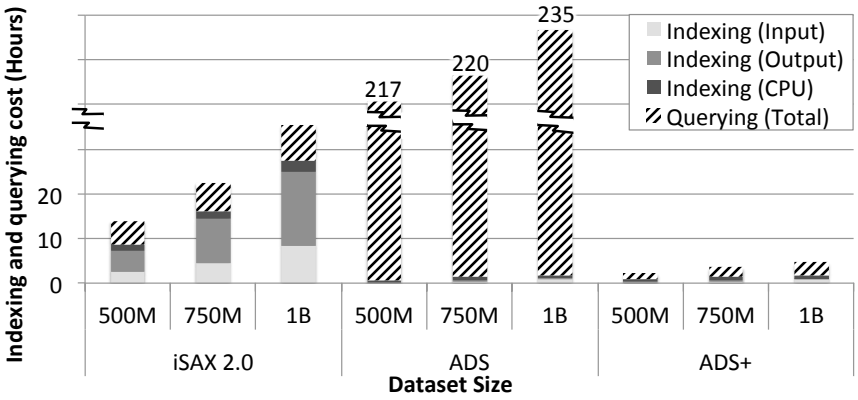
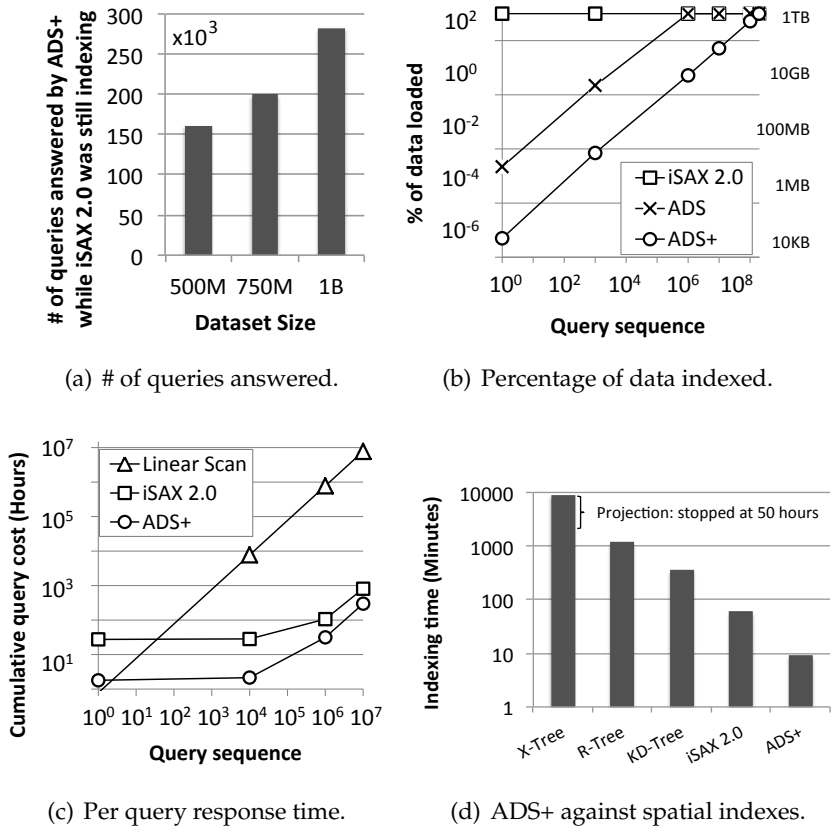



Figure 3.9: Scaling to 1 billion data series.

*For 1 billion data series, ADS+ answers all* $10^5$ *queries in less than 5 hours, while iSAX 2.0 needs more than 35 hours.*

By adaptively expanding the tree and adjusting leaf sizes only for the hot workload parts, ADS+ enjoys a $7x$ gain over full indexing in iSAX 2.0. In addition, ADS+ significantly outperforms ADS; even though ADS can significantly reduce indexing costs for all data sizes, as we process more and more queries it suffers due to the high cost of fetching unindexed data series for large leaves during query processing. ADS+ avoids this problem by adaptively splitting its leaves. Also, the rate at which the cost of ADS+ grows is significantly smaller than that of iSAX 2.0; For example, going from 500M to 1B data series, iSAX 2.0 needs more than twice the time, while ADS+ enjoys a




(a) # of queries answered.

(b) Percentage of data indexed.

(c) Per query response time.

(d) ADS+ against spatial indexes.

Figure 3.10: Reducing the data-to-query time with ADS+ as we scale to big data.

sub-linear cost increase.

Figure 3.10 provides further insights. Figure 3.10(a) depicts the number of queries that ADS+ can answer within the time that iSAX 2.0 is still indexing. The bigger the data set, the more queries ADS+ can answer before iSAX 2.0 answers even a single query; for the case of 1 Billion data series ADS+ manages to answer nearly $3 * 10^5$ queries while iSAX 2.0 is still indexing. this verifies the fact that ADS+ is more suited towards very large data sets compared to traditional non-adaptive indexing approaches.

**Data Touched.** In addition, Figure 3.10(b) shows the amount of data actually touched (indexed) as the query sequence evolves. To see the long term effect, we let a big number of queries run, i.e., $10^7$ queries. For iSAX 2.0 the behavior in Figure 3.10(b) is a flat curve as everything is indexed blindly up front. With ADS and ADS+ though, we index a much smaller percentage of the data; as more queries are processed, more data is indexed and only when needed. While ADS indexes all data by the time it processes $10^6$ queries, ADS+ manages to touch even less data; since it splits leaves adaptively to

much smaller sizes it needs to materialize much smaller leaves and thus it touches less data overall. In this way, even after $10^7$ queries it has touched only 10% of the data, while it needs more than 190M queries in order to touch all the data (i.e., completely build the index). In fact, since this is a random workload, this is the worst case for adaptive indexing as most queries lead to fetching raw data series and enriching the index. This is why ADS has touched all data by query $10^6$; most queries will need to materialize a partial leaf and thus they need to fetch $2 * 10^3$ new data series (its leaf size); $2 * 10^3 * 10^6$ adds up to well above $10^9$ (the data set size). On the contrary, ADS+ uses a query-time adjustable leaf size of only 10 data series; thus even if all queries need to fetch new data, by query $10^7$ we would have fetched at most $10^8$ data series which is about the 10% (of the original $10^9$ data set) we see in Figure 3.10(b). By doing less work and only when necessary, ADS+ allows users to have quick access to their data.

**Per Query Performance.** We continue our study with a discussion that focuses on the individual query performance based on the previous 1 Billion data series experiment and 10 Million random queries. Here we also include the scan strategy, i.e., when we do not build an index; instead, every query performs a complete scan over all data series. We will not use ADS from now on as ADS+ consistently outperforms ADS.

Figure 3.10(c) shows the cumulative per query response time as the query sequence evolves. The scan strategy has a constant but slow response time; every query adds the same cost to the total cumulative costs. Eventually, the scan strategy becomes prohibitive if we want to repeatedly query the same big data; it takes close to $10^5$ hours to handle all queries. iSAX 2.0 pays a big cost to build the index (this is included in the cost of Query 1) but then queries are very fast, i.e., the cumulative cost curve is flat as every query adds very little cost. Once the index is built, every iSAX 2.0 query incurs a constant cost; still though there is a big bottleneck to access the data due to the high indexing costs which means that the first query needs to wait for several hours. On the contrary, ADS+ enjoys quick data access time; it finishes building the index and answering all queries by the time iSAX 2.0 is still indexing and has not answered a single query.

In fact, while the crossover point of the scan strategy with iSAX 2.0 is at about 35 queries, for ADS+ it is only at 2 queries. This means that for iSAX 2.0 to be useful we need to fire at least 35 queries while ADS+ starts bringing gains already after the first 2 queries. Moreover, while the average query answering time for ADS+ is about 50 milliseconds, that of iSAX 2.0 is 200 milliseconds. In other words, iSAX 2.0 is never going to amortize its initialization overhead over ADS+ and thus it is always beneficial

to use adaptive indexing as opposed to full a priori indexing. This is because of the larger leaf size that is used by iSAX 2.0, in order to reduce the index building time by compromising query times a bit. On the other hand, ADS+ adaptively splits leaves for the hot part of the data and thus it can reduce access times even further. Furthermore, the cost of query answering for ADS+ (essentially, materializing the data of the leaf) increases linearly with leaf utilization. This cost ranges from 20ms when the leaf is already materialized to 160ms when the leaf contains all 10 data-series that need to be loaded from the raw file. When ADS+ needs to perform splits, the query answering times are 129ms for 1-10 splits, 138ms for 10-20 splits, 148ms for 20-30 splits, and 160ms for 30-40 splits. All these times are significantly smaller than the required time to answer a query using serial scan (more than 46min).

### 3.4.2 ADS+ vs. Multi-dimensional Indexes

One interesting question is how indexes which are tailored for data series search compare against state-of-the-art spatial indexes. In this experiment, we compare ADS+ and iSAX 2.0 against KD-Tree [18], R-Tree [68], and X-Tree [19], a state-of-the-art adaptive version of R-Tree. X-Tree creates a tree with minimal overlap between nodes and it allows for variable sized nodes in order to accommodate minimum overlapping. Such spatial indexes can be used for indexing data series and performing similarity search; the main idea is that we can use the PAA representations of data series to create a KD-Tree, an R-Tree, or an X-Tree.

Here, we use a set of 100 million data series. In all the cases, the amount of dimensions for the reduced dimensionality PAA representation is set to 16 while the original size of each data series is 256 points. Figure 3.10(d) depicts the time needed to complete the index building phase for each index. Overall, both data series tailored indexes, iSAX 2.0 and ADS+, significantly outperform the more generic spatial indexes. For example, iSAX 2.0 is one order of magnitude faster than R-Tree while ADS+ is two orders of magnitude faster, and more than an order of magnitude faster than KD-Tree. The raw benefit comes from the fact iSAX 2.0 and ADS+ are tailored to perform efficient comparisons of SAX representations (with bitwise operations). ADS+ being adaptive enjoys further benefits as we discuss in previous experiments as well. X-Tree is significantly slower as a result of its more expensive index building phase which focuses on minimizing overlap between nodes. Naturally, this helps query processing times as less overlap allows queries to focus faster on data of interest. However, as we have shown throughout the analysis in this paper, as we scale to big data, index building is

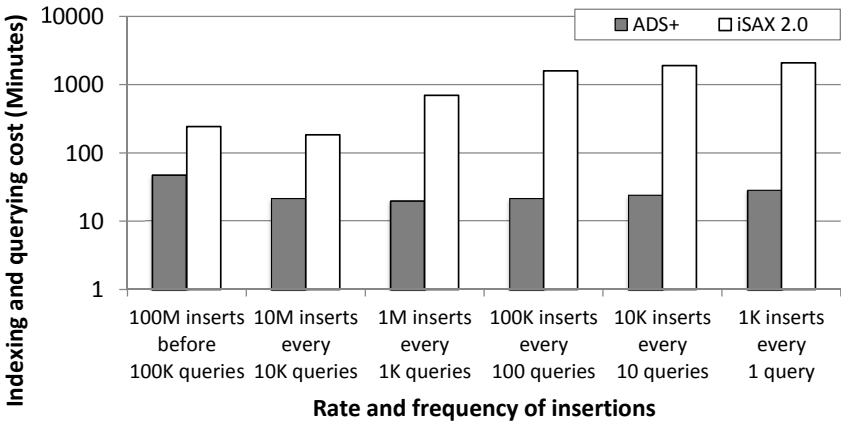


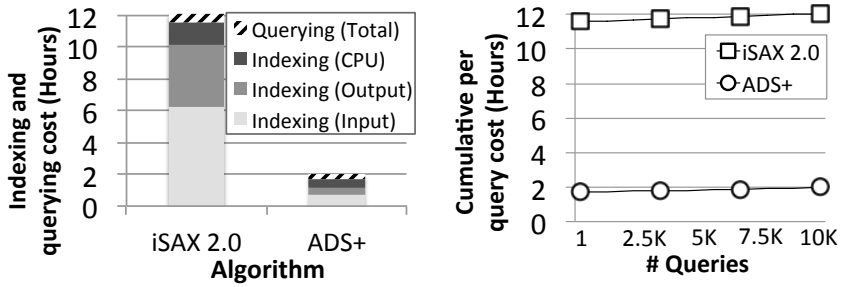

Figure 3.11: Updates for 100 million data series and 100 thousand queries in 6 different batch sizes.

the main bottleneck and thus X-Tree is prohibitively expensive.

### 3.4.3 Adaptive Behavior under Updates

In our next experiment we study the behavior of ADS+ and iSAX 2.0 with updates. We use a synthetic data set of 100 million data series and $10^5$ random queries. This time, queries are interleaved with updates. In particular, we perform the experiment in 6 steps. Each time a varying number of new data series arrive and at different query intervals. Figure 3.11 shows the results. The first set of bars represents the case where all data has arrived up front and all queries run afterwards. The second set of bars (10M inserts every 10K queries) represents a scenario where every $10^4$ queries $10^7$ new data series arrive until we reach a total of $10^8$ data series (i.e., the complete data set) and a total of $10^5$ queries (i.e, the complete query workload). Similarly, the rest of the bars vary the frequency and the rate of incoming data until the extreme case where we get 1000 new insertions after every single query.

In all cases, ADS+ maintains its drastic performance advantage over iSAX 2.0. When all data arrives up front, the cost is naturally higher; more data has to be queried. For the rest of the cases where data arrives incrementally, interleaving with queries, we observe that when data arrives more frequently the overall cost increases slightly. This is a result of both the fact that merging of updates needs to happen more often and of the fact that more queries need to be processed against more data. However, even in the extreme case where we receive 1000 new data series after every query, ADS+




(a) Total cost.

(b) Cumulative per query costs (Query 1 includes indexing).

Figure 3.12: (TEXMEX) Indexing 1 Billion images (SIFT vectors) and answering $10^4$ queries.

maintains its adaptive behavior and good performance being able to outperform static iSAX 2.0 by 2 orders of magnitude.

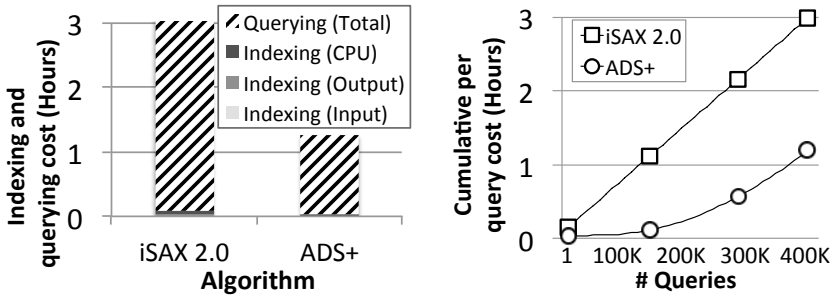
The behavior under deletions is similar. For example in experiments with a data set of 100 million data series, indexed by ADS+, we could perform deletions with an average deletion time at 0.2 milliseconds.

### 3.4.4 Reducing the Data-to-query Time in Real-life Workloads

Here, we demonstrate the ability of ADS+ to drastically reduce the data-to-query time in real-life scenarios with real data. In all cases, we use the optimal settings found in the synthetic benchmarks: for iSAX 2.0 uses a leaf size of 20K data series, while ADS+ uses a build time leaf size of 2K data series which adaptively drops down to 10.

**Texmex Corpus** (TEXMEX). The first real-life scenario is an image analysis scenario from the Texmex corpus [84]. This dataset contains 1 Billion images which are translated into a set of 1 Billion data series (SIFT feature vectors) of 128 points each. The scenario is that a user is searching the corpus for images similar to an existing image that they already have. The corpus also contains $10^4$ such example queries together with information about which image in the corpus is the nearest neighbor, i.e., the most similar one, for each query.

Figure 3.12 shows the results. Figure 3.12(a) shows the total cost to go through the indexing phase and to process all queries. ADS+ maintains its drastic gains as we have seen in the synthetic benchmarks study. Overall, ADS+ finishes answering all queries 6 times faster compared to iSAX 2.0. It is interesting to mention that ADS+ gains not only during the indexing phase but also during the query processing phase, i.e., the time it takes to answer all $10^4$ queries is smaller with ADS+. This is because these real-life




(a) Total cost.

(b) Cumulative per query costs (Query 1 includes indexing).

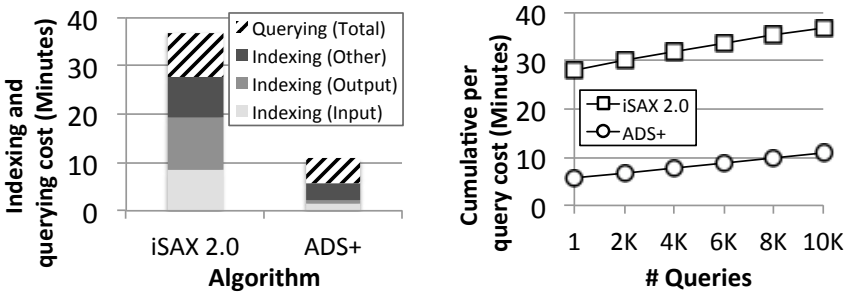
Figure 3.13: (DNA) Indexing 20 Million DNA subsequences from the Homo Sapiens genome and answering $4 * 10^5$ queries.

queries are not completely random, i.e., the workload focuses in specific areas of the index. In such cases, ADS+ has the benefit of working on an index which essentially contains less data; it has loaded only the data which are relevant for the hot workload set.

Figure 3.12(b) helps to understand this behavior even more by demonstrating the evolution of the query processing costs, i.e., the graph shows how the indexing and query processing costs evolve through the query sequence for each indexing strategy. For iSAX 2.0 the first query needs to wait until the whole index is built which takes almost 12 hours. From there on, each query can be processed quite fast. On the contrary, ADS+ allows the first query to access the data in less than 2 hours, while by the time we reach the 2 hours mark all $10^4$ queries have been processed. Overall, ADS+ process all queries in just 2 hours, while iSAX 2.0 needs more than 11 hours just for the indexing phase and without processing a single query.

**DNA Data** (DNA). The second real-life scenario comes from the biology domain. This dataset contains the full genome of the Homo Sapiens (human) which is translated into 20 Million data series of 640 points each, obtained using a sliding window of size 16000, down-sampled by a factor of 25. The scenario is that a user is trying to identify subsequences of the human genome that match subsequences in other genomes. In this way, we create our queries from the genome of the Rhesus Macaque ape which is also translated into 20 Million data series of 640 points each, obtained in the same manner, and each one of these data series can be used as a query against the human genome in search for similar patterns.

Figure 3.13 shows the results. Similarly to previous experiments, ADS+ brings a sig-




(a) Total cost.

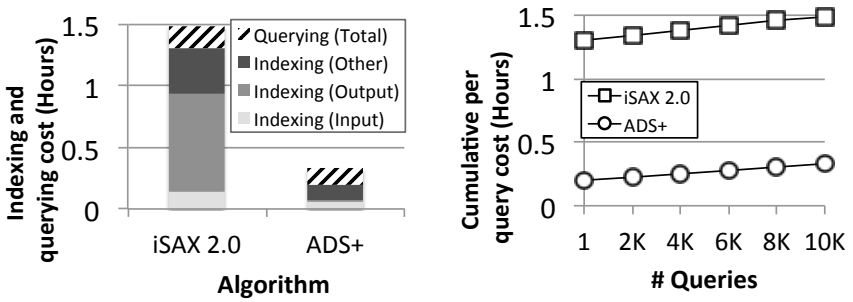
(b) Cumulative per query costs (Query 1 includes indexing).

Figure 3.14: (SEISMIC) Indexing 100 Million seismic data series and answering $10^4$ queries.

nificant benefit both in terms of total costs and in terms of per query costs. With ADS+ we can index the data and process all queries 3 times faster, i.e., only after one hour, while with iSAX 2.0 we need to wait for 3 hours. Compared to previous performance examples, it is interesting to note that in this experiment we have a very different data to queries ratio, i.e., we have a relatively small data set of 20 Million data series and a relatively big query set of $4 * 10^5$ queries. Thus, the indexing cost is a much smaller factor of the total cost compared to previous experiments. Still though, ADS+ brings a major benefit and shows a scalable behavior, mainly due to its ability to adapt its shape to workload patterns, by expanding sub-trees and adjusting leaf sizes on-the-fly.

**Seismic Data** (SEISMIC). The third real life scenario is one that comes from seismology. We used the IRIS Seismic Data Access repository [8] to gather data series representing seismic waves from various locations. We obtained 100 million data series of size 256 using a sliding window with a resolution of 1 sample per second, sliding every 4 seconds. The complete dataset size was 100GB. We additionally obtained 10,000 data series with the same technique to be used as queries. We used iSAX 2.0 and ADS+ to index the data and answer all the queries in the workload. Figure 3.14 shows the results. ADS+ can index the data more than 4 times faster than iSAX 2.0. With ADS+ we need to wait just under 6 minutes before we fire our first query, while iSAX 2.0 needs more than 25 minutes. In regards to query answering, we are able to index the data and answer all the $10^4$ queries in 11 minutes with ADS+, while iSAX 2.0 requires 37 minutes to complete the same task.

**Astronomical Data** (ASTRO) In the last real scenario, we used astronomical data series representing celestial objects [168]. The dataset comprised of 200 million data series of size 256, obtained using a sliding window with a step of 1. The total dataset




(a) Total cost.

(b) Cumulative per query costs (Query 1 includes indexing).

Figure 3.15: (ASTRO) Indexing 200 Million astronomical data series and answering $10^4$ queries.

| Workload | Cross-over point (PADS+ over ADS+) |
|---|---|
| Random | 2899 queries |
| Low skew | 2970 queries |
| Medium skew | 3097 queries |
| High skew | 3825 queries |

Table 3.2: Fast access with PADS+ with varying skew.

size was 200GB. We obtained an additional 10,000 data series from the raw dataset using the same technique to be used as a query workload, and used both iSAX 2.0 and ADS+ to answer the complete workload. Figure 3.15 shows the results. In this case, ADS+ is more than 6 times faster in indexing time than iSAX 2.0. With ADS+ we need to wait about 12 minutes before we fire our first query, while iSAX 2.0 needs more than 75 minutes. In regards to query answering, we are able to index the data and answer all $10^4$ queries in less than 20 minutes with ADS+, while iSAX 2.0 requires 1.5 hours.

### 3.4.5 Providing Quick Insights with PADS+

Having shown that it is possible to reduce the user waiting time, without excessively penalizing the query answering time, we now show that we can achieve even faster access to the data for skewed workloads. In this experiment we analyze the performance of ADS+, PADS+ and iSAX 2.0 over a dataset of 1 billion data series and a varying set of query workloads, ranging from completely skewed to completely random queries. In total, we run $10^4$ queries. For low skew, 60% of the queries are picked from 40% of the domain. In the medium skew workload, 80% of the queries are picked from 20% of

the domain, while for the high skew workload 99.99% of the queries are picked from 0.01% of the domain.

For all workloads both ADS+ and PADS+ significantly outperform iSAX 2.0 being 10 to 20 times faster. iSAX 2.0 needs about 28 hours to index all data and process all queries with the bulk of the time spent in indexing (included in the cost of Query 1). Both ADS+ and PADS+ can do so in less than 1.5 hours for $10^3$ queries, and less than 3 hours for $10^4$ queries. ADS+ improves slightly as skew increases; less data has to be fetched from outside the index. PADS+, though, as seen in Table 3.2, manages to improve performance even more as skew increases, being faster than ADS+ and iSAX 2.0 for all skewness levels for the first 2000 queries and for almost 4000 queries in the case of high skewness. When the workload is skewed, this means that PADS+ can focus on certain parts of the index tree and avoid node splits and disk spilling once it optimizes the index for the hot part.

While ADS+ provides the best overall solution being both fast and robust, PADS+ provides an attractive solution when we know we want to fire only a few thousands of queries.

## 3.5 Summary

In this chapter, we showed that state-of-the-art data series indexing approaches cannot cope with the data deluge. The time needed to build a data series index becomes prohibitive as the data grows, and may take more than 24 hours to index a collection of 1 billion data series. We proposed an adaptive indexing approach, where the index is built incrementally and adaptively, resulting in a very fast initialization process. Both the shape of the tree index and the leaf sizes are tuned adaptively and automatically to fit the workload on-the-fly. Using both synthetic and diverse real-life data, we show that our new adaptive indexing approach copes significantly better with the ever growing data series collections, and can answer several thousands of queries in the time that state-of-the-art indexing approaches are still in the indexing phase.

# Chapter 4

# Minimizing the Query to Answer Gap for Adaptive Data Series Indexing

## 4.1 Introduction

Approximate search on the iSAX family of indexes (of which ADS is a part of) has been shown to be able to retrieve high quality nearest neighbors. Nevertheless, while ADS+ is able to outperform all state-of-the-art solutions in approximate query answering, it suffers heavily in the case where an exact answer is required. This is because of the need of materializing multiple leaves during query answering. This is especially true in cases where queries are extremely difficult, and the index is unable to perform any pruning. In such cases, ADS might need to materialize the complete index in just a single query. In order to avoid this problem, there are two courses of action, either full indexing or developing novel exact query answering algorithms.

### 4.1.1 Contributions

In this chapter, we study both the aforementioned directions and provide a new full index construction algorithm, as well as a new exact query answering algorithm, both outperforming previous work.

#### Efficient Full Index Construction with ADS-Full

If a highly uniform and large workload is expected, and exact queries are required, one could consider constructing the complete data series index beforehand. This is because the more queries are answered and the more uniform they are, the larger the

percentage of the index that we will have to construct during adaptation will be. Nevertheless, as we've seen in the previous subsection, full index construction can be really time consuming. To overcome this problem, we propose a novel full index construction method, based on ADS, which outperforms state-of-the-art indexing techniques, constructing the exact same tree with much less time overhead. Since very large data series collections cannot completely fit in main memory, they have to be read in batches. Our observation in this process, is that the most significant cost during traditional full index construction is incurred by the action of writing intermediate results on disk and performing split operations on them, which are mostly random I/O. Such random disk operations dramatically decrease the performance of index construction algorithms. Our approach can construct the complete index using just two sequential passes over the complete dataset. Our key intuition is that while the raw data series cannot fit in main memory, their summaries can, and that the complete index structure can be built just using them. The raw data are then placed at the right leaves using a second sequential pass over the complete data set. Our method, called ADS-Full, is presented in detail in Section 4.2, and is shown to be 40% faster than the state-of-the-art.

#### Efficient Exact Search for ADS+ with SIMS

When a very large and uniform workload is not expected, but we still need to answer exact queries, we need a way to contain the cost of query answering. This means that we need a way of controlling the percentage of the index that is materialized during each query. Traditional exact search algorithms proceed to check an index node at a time, sorting them in a most promising first fashion. While this case is a efficient in full indexes, this is not the case for an adaptive index. This is because each node visited is a node materialized. A process which includes visiting the raw data file and moving data to the right position on the index. As a result, when queries need to check a very large amount of nodes, the cost would be prohibitive for adaptive indexing. To contain this cost we propose a novel algorithm for exact search called SIMS. The key intuition is that a summarized version of the complete data set can fit in main memory. These summaries can be used to perform pruning. We start the algorithm with an approximate search, thus materializing only a fixed number of nodes at a time, and then use the summaries in memory to prune data in the raw file. The summaries are aligned to the raw data file and only data that cannot be pruned are accessed. We describe our algorithm in detail in Section 4.3.

## 4.2 Creating Fully Materialized Indexes with ADS-Full

In settings where a complete index is required, i.e., when there is a completely random and very large workload, a full index can also be efficiently constructed with ADS. Our approach, called *ADS-Full*, is comprised of two steps. In the first step, the ADS structure is built by performing a full pass over the raw data file, storing only the iSAX representations at each leaf. In the second step, one more sequential pass over the raw data file is performed, and data series are moved in the correct pages on disk.

The benefit of this process is that it completely skips costly split operations on raw data series: indeed, split operations are performed only on iSAX summarizations, and mostly within the bounds of main memory. The reason is that iSAX summarizations correspond merely to 1.5% of the raw data size, and as a result 1 TB of raw data series can be summarized with 16 GB using iSAX summarizations. This means that a single pass over the raw data file enables the construction of the complete index using the iSAX summaries, entirely in main-memory. In this case, all split operations are performed in main memory, and the data structure is flushed on disk only after the entire process has finished.

During the second step, the raw data file is read again, and their appropriate locations on disk are identified by index lookup operations, as follows: we compare the isax summary of each raw series to the ADS-Full nodes during a single path traversal of the index, until we identify the leaf node in which this series belongs in. These are mostly binary operations, and as a result, extremely fast. Data are then buffered at the LBL level, and when there is no more free main memory, they are sequentially flushed on disk. As we demonstrate in the experiments section, this approach is 40% faster than building the complete index using iSAX 2.0.

## 4.3 The SIMS Exact Search Algorithm for ADS+

Approximate Search in ADS+ (Algorithm 9) works by visiting the single most promising leaf, and calculating the minimum distance to the raw data series contained in it. This allows us to provide an approximate solution that is close to the actual answer, while at the same time controlling the time spent on reading raw data from the index.

Exact Search on the other hand, requires visiting a much larger part of the dataset, in order to guarantee that the returned answer is truly the closest match to the query in the entire collection. Traditionally, such algorithms (like Algorithm 28 for ADS+) push index nodes into a priority queue, based on their minimum distance estimation. The

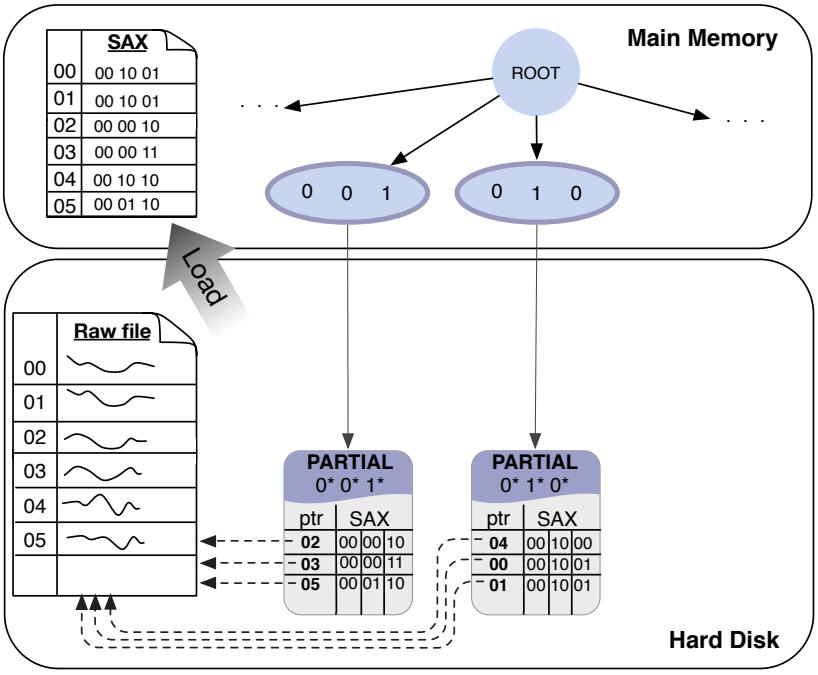


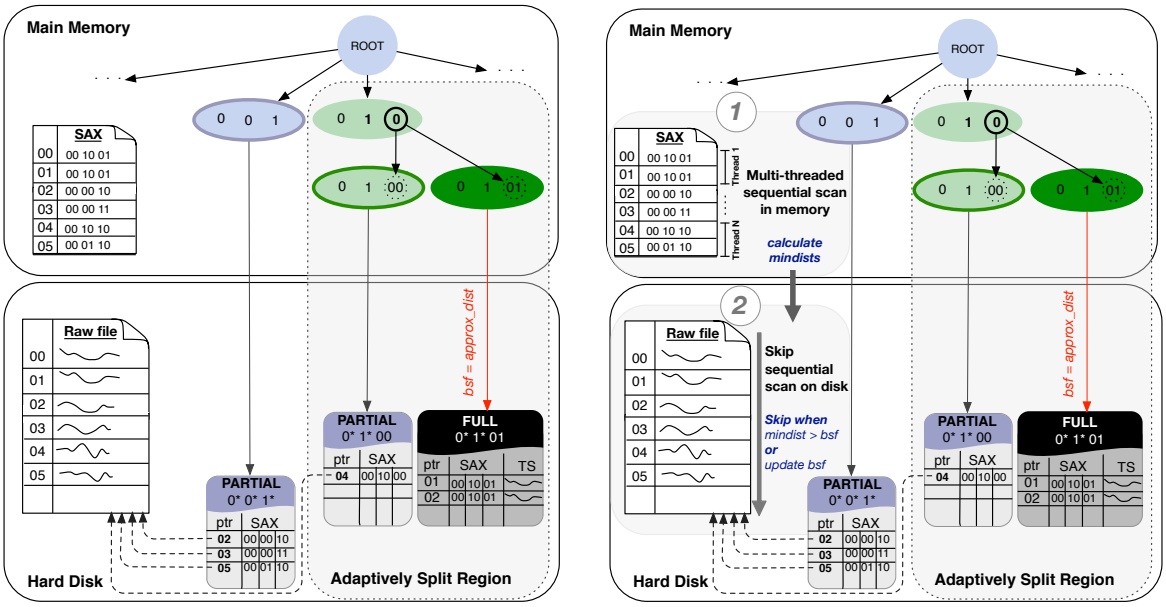

Figure 4.1: SIMS initial state.

"closest" ones are the nodes visited first, and the answer is gradually refined as more leaves are visited.

While a large number of raw data may be pruned, the disk accesses involved in this process are random. This is because the raw data-series for each leaf reside on a different page of the disk, and leaves are visited in a most-promising-first fashion. In this way, a significant number of CPU cycles is wasted waiting for data to be fetched from disk.

To overcome this problem, we propose a skip sequential scan algorithm: it employs approximate search as a first step in order to prune the search space, it then accesses the data in a sequential manner, and finally it produces an exact, correct answer. We call this algorithm Scan of In-Memory Summarizations (SIMS). The main intuition is that while the raw data do not fit in main memory, their summarized representations, which can be orders of magnitude smaller, will fit. For example, the size of a 16-segment iSAX representation for a single data series is 16 bytes, while a raw data-series of 256 float points is 1,024 bytes. The iSAX summaries of 1 billion data series occupy merely 16GB in main memory. By keeping these data in-memory and scanning them, we can estimate a bound for every single data series in the dataset.

The algorithm (refer to Algorithm 10) starts by checking if the SAX data are in mem-




(a) ADS+ Approximate Search.

(b) Skip-sequential Scan using SAX.

Figure 4.2: SIMS during query answering.

ory (lines 2-3), and if not it loads them. It then proceeds to create an initial best-so-far (BSF) answer (line 5), using the Approximate Search algorithm of ADS+ (Figure 4.2(a)). A minimum distance estimation is performed between the query and each in-memory SAX record (lines 7-10), using multiple parallel threads, operating on different subsets of the data. For each lower bound distance estimation, if it is smaller than the real distance to the BSF, we fetch the complete data series from the raw data file and calculate the real distance (lines 12-14). If the real distance is again smaller than the BSF, we update the BSF value (lines 15-16).

Since the summaries array is aligned to the data on disk, what we essentially do is a synchronized skip sequential scan of the raw (on-disk) data and the (in-memory) mindists array. This property allows us to prune a large amount of data, while ensuring that we do sequential reads in both main memory and on disk, as well as enable modern multi-core CPUs to operate in parallel on the data (the SAX summaries in this case) stored in main memory. The algorithm finally returns the final BSF to the user, which is the exact answer to the query.

The initial state of the index is depicted in Figure 4.1, where the SAX data can be seen alongside the index in main memory. Initially, SIMS performs an Approximate Search operation, performing adaptive splits and loading data from the raw file as nec-

**Algorithm 10:** exactSearchSIMS(**dataSeries**, **isax**, **index**, **queryTimeLeafSize**, **file**)

```
1  // If SAX summaries are not in-memory, load them
2  if SAXSummarizations = ∅ then
3  |   SAXSummarizations = loadSAXFromDisk();
4  // Perform an approximate search
5  bsf = approxSearchADS+(dataSeries, isax, index, queryTimeLeafSize);
6  // Compute minimum distances for all summaries
7  Initialize mindists[] array;
8  // Start multiple threads & compute bounds in parallel parallelMinDistsCompute(mindists,
   SAXSummarizations, dataSeries);
9  // Read raw data for unprunable records recordPosition = 0;
10 for mindist ∈ mindists do
11 |   if mindist < bsf then
12 |   |   Move file pointer to recordPosition;
13 |   |   rawData = read raw data series from file;
14 |   |   realDist = Dist(rawData, dataSeries);
15 |   |   if realDist < bsf then
16 |   |   |   bsf = realDist;
17 |   recordPosition++;
18 return bsf
```

essary. This can be seen in Figure 4.2(a). Given a BSF solution produced by Approximate Search a multi-threaded process computes the lower bounds to all in-memory summarizations and a skip-sequential read of the raw file is performed. This is shown in Figure 4.2(b).

It is important to notice, that SIMS works well even in the degenerate case where our dataset comprises of identical data series. In such a case, even a single exact query could lead to the materialization of the complete index. SIMS avoids this situation, since the rate with which data are materialized is fixed across all queries, and data loading happens only during the approximate part of the algorithm.

## 4.4 Complexity Analysis

We now provide a space complexity analysis for ADS, as well as a time complexity analysis for all the search algorithms we have presented. Since the actual size of the

index as well as the time needed to answer each query highly depends on the data distribution [199], we concentrate in providing lower and upper bounds for indexing and query answering.

**Best Case.** The ADS index is the most compact when (after all adaptive split operations) it has the smallest possible number of nodes, and all the leaf nodes are completely full. If we have $N$ data series, and all leaves are full, then we have a total of $l_{min} = \lceil \frac{N}{th} \rceil$ leaves, where $th$ is the query-time leaf size. In order to have the shortest possible tree, every level of the tree must have the highest possible fan-out. If $w$ is the number of iSAX segments used, the root node of ADS has $2^w$ children that form binary trees. In the best case we have one binary tree for every single root child, with $\lceil 2(\frac{l_{min}}{2^w}) - 1 \rceil$ inner (in-memory) nodes, and $\frac{l_{min}}{2^w}$ leaf (on-disk) nodes each. In total, the smallest possible ADS index will have $n_{min} = 1 + 2^w \left\lceil \left( \frac{\lceil \frac{N}{th} \rceil}{2^{w-1}} \right) - 1 \right\rceil$ nodes (1 root node, and $2^w$ full binary trees with $\lceil \frac{N}{th} \rceil$ leaves equally distributed among them).

Approximate search in this case requires the traversal of a single path from the root of the index to one of the leaves. This is $\Theta\left(log_2\left(\left\lceil \frac{\lceil \frac{N}{th} \rceil}{2^w} \right\rceil\right)\right)$ in-memory accesses and 1 disk read of size $\Theta(th)$ ($th << N$).

**Worst Case.** In the worst case, all data series in the ADS index end up in just one of the root children nodes, and all subsequent split operations are unable to separate the data. This would happen only if all the data series were almost identical (in which case using an index would be pointless anyways), and would result in an index with one single leaf (with leaf size $th_{expanded} = N$). The maximum length of the path to this leaf depends on the number of split operations. For $w$ iSAX segments and $s$ bits per segment, the maximum number of split operations is then $w(s-1)$. Approximate search in this case would require $\Theta(w(s-1))$ in-memory accesses and 1 disk read of size $\Theta(N)$.

**Exact Search.** In the best case, exact search will need to pay the cost of one approximate search and $2^w$ in-memory accesses for retrieving and pruning all the root level children. This is a constant number of in-memory accesses above approximate search.

In the worst case, exact search will access the entire index structure using random disk accesses. Things are different in the case of SIMS, where it is ensured that all disk accesses are sequential. In the best case, SIMS will do just one approximate search and one complete scan over all the iSAX summarizations. In a typical setting, this should be around 1.5% of the raw data size. In the worst case, SIMS will additionally need to perform one full sequential pass over the raw data file as well.

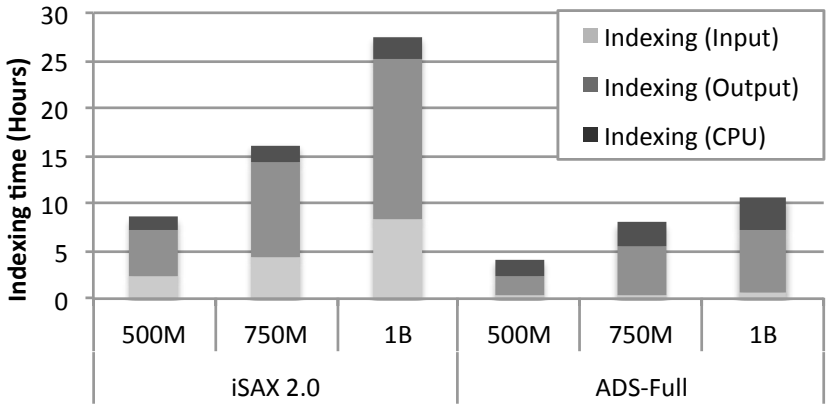



Figure 4.3: ADS-Full constructs the complete index in 38% of the time that iSAX 2.0 requires for 1B data series.

## 4.5 Experimental Evaluation

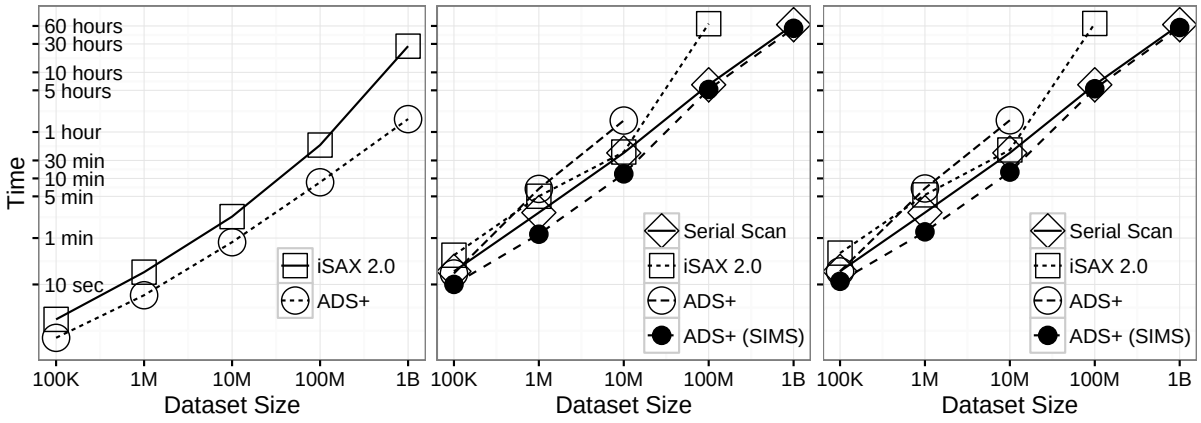
### 4.5.1 Fast Full Index Construction

In this subsection, we demonstrate the benefits of ADS-Full for index initialization even when building the whole index in one step. For this experiment, we indexed 500M, 750M and 1B randomwalk generated data series of size 256, and compared ADS-Full with iSAX 2.0. Note that the indexes created by both ADS-Full and iSAX 2.0 are exactly the same: they contain the same inner nodes, and the same leaf nodes (along with the same corresponding raw data series). Following our earlier discussion, for both iSAX 2.0 and ADS-Full we used a leaf size of 20K.

The results are depicted in Figure 4.3. We observe that for 500M and 750M data series, ADS-Full requires 48% of the time of iSAX 2.0 in order to build the full index, while for the case of 1B data series ADS-Full completes the task in just 38% of the time required by iSAX 2.0. These results demonstrate that our approach outperforms the state of the art, even for the task of building a full index, for which iSAX 2.0 was initially designed.

### 4.5.2 Efficient Exact Query Answering using SIMS

**Setup.** In this subsection, we explore the benefits of using SIMS for answering exact queries. We use both ADS+ and iSAX 2.0 to index 5 random walk generated datasets with sizes of 100K, 1M, 10M, 100M, and 1B data series of length 256. Each data series has a record size of 1024 bytes. We generate queries by adding Gaussian noise to randomly selected data-series from the original dataset. The more noise we add,




(a) Indexing times. (b) Querying times. (c) Indexing and querying times combined.

Figure 4.4: Indexing 1 billion data series and issuing 100 exact queries.

the harder the queries become, as they drift away from their nearest neighbor. We use queries with varying amounts of noise, in order to test the algorithms under different conditions. For each dataset we generate 100 queries, and use the following four methods to answer them.

- **Serial Scan**. This is a baseline approach, which has been shown to outperform the exact search of iSAX 2.0 in several cases [90]. We answer each query by performing a full sequential scan of the raw data file. This method implements the early abandoning technique, where we stop scanning and evaluating the distance for a data series when this distance becomes greater than the best-so-far solution.

- **iSAX 2.0**. This is the exact search algorithm of iSAX 2.0. We use the complete iSAX 2.0 index that we have built beforehand and visit nodes in a most-promising-first fashion. All nodes are pushed in a queue and the one with the minimum lower bound is popped. If this node is a leaf, then we check the full data series, retrieved from disk.

- **ADS+**. This is ADS+ implementing the exact search algorithm of iSAX 2.0. The only difference is that when we visit leaf nodes we first perform adaptive split operations and then load the data from the raw data file in the index.

- **ADS+ (SIMS).** This is the SIMS exact search algorithm. We load all the iSAX representations in main memory and perform a multi-threaded lower bound calculation. We then visit the data on the raw file for only the records with a lower

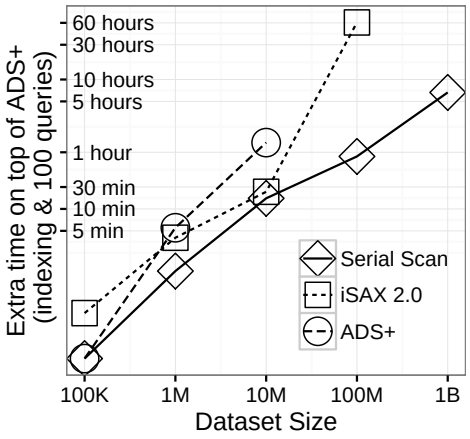



Figure 4.5: Extra time on top of ADS+ (with SIMS) for all other methods.

| Dataset size | Cross-over point (Serial Scan over ADS+) |
|---|---|
| 100K | 7 queries |
| 1M | 4 queries |
| 10M | 4 queries |
| 100M | 3 queries |
| 1B | 3 queries |

Table 4.1: ADS+ outperforms serial scan after a few queries.

bound less than the best-so-far solution obtained using Approximate Search.

We have removed the square root computation from the Euclidean Distance, for all the above approaches.

**Evaluation.** In Figure 4.4(a), we plot the indexing time for both iSAX 2.0 and ADS+. Serial Scan has no initialization cost. ADS+ outperforms iSAX 2.0 by more than an order of magnitude in terms of data-to-query time. In Figure 4.4(b), we plot the query answering time for all algorithms, and in Figure 4.4(c), we plot the indexing and query answering times combined, when the dataset varies between 100K and 1B data series.

ADS+ (SIMS) is the fastest method across the board. The speed-up is more pronounced when the complete dataset fits in main-memory, where we are able to prune at a per data series level. This is because data are transferred from main memory to the CPU in cache-lines, which are much smaller than the data series size. Consequently, we have fine control over the data series that are transferred to the CPU: these are only the data series that need to be processed. On the contrary, in the case of large dataset sizes

that exceed the main memory capacity (i.e., 10M and above in our experiments), we are only able to prune at a per disk-page level. Since disk pages fit more than one data series, we end up wasting a considerable amount of time on reading and transferring from disk data series that are not needed.

We observe that the benefit of ADS+ (SIMS) in absolute numbers increases with the dataset size. This is depicted in Figure 4.5, where we plot the amount of additional time that all methods need in order to (index the dataset and) answer all the queries in the workload, when compared to ADS+ (SIMS). For the 10M dataset, Serial Scan, iSAX 2.0, and ADS+ respectively need 2.1x, 2.4x, and 7.4x more time than ADS+ (SIMS), which completes the task within 12.7 minutes. For the 1B dataset, Serial Scan needs 7 hours more than ADS+ (SIMS) in order to produce the results; iSAX 2.0 and ADS+ required more than 60 extra hours, at which point we stopped their execution.

Our experimental evaluation also shows that, even if Serial Scan has zero initialization cost, ADS+ (SIMS) very quickly outperforms it, after only a few queries (refer to Table 4.1). With a dataset of 100K data series, Serial Scan becomes slower than ADS+ (SIMS) if we want to answer 7, or more, queries. Moreover, the relative benefit of ADS+ (SIMS) increases with the dataset size: for the datasets with more 100M data series, ADS+ (SIMS) is faster than Serial Scan after answering merely 3 queries. As a result, ADS+ is the best option in all cases, even when analysts need to answer only a few queries.

According to our complexity analysis of Section 4.4, being able to answer queries faster than the Serial Scan, when including the indexing cost as well, means that we are far away from the worst case scenario, efficiently pruning large parts of the raw dataset.

## 4.6 Summary

In this chapter we presented algorithms that minimize the query answering time for ADS+. We developed a full index construction method based on ADS, which outperforms the state-of-the-art, and a novel exact query answering algorithm that facilitates sequential disk scans in order to speed up exact search. In conclusion, we show that when data are stored in traditional disks, doing partial serial scans on scattered data is always much more efficient than doing random access. This is also the case with our new index construction algorithm, which instead of writing intermediate results, it relies on a double serial pass of the raw dataset.

# Chapter 5

# The RINSE Data Exploration System

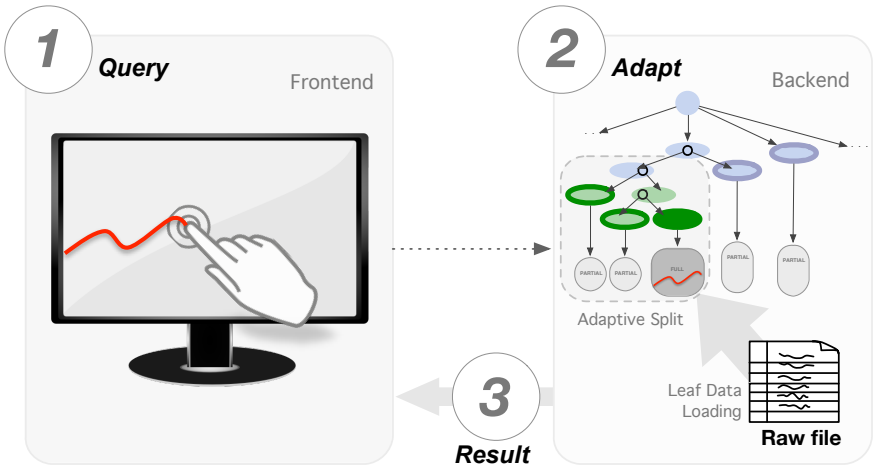



Figure 5.1: Interactive Data Series Exploration

In this chapter we present the RINSE system (a recursive acronym: RINSE Interactive Series Explorer), which is built around ADS+. RINSE provides a user interface that manifests the benefits of adaptive indexing for interactive exploration of large data series collections. The exploration process can be seen in Figure 5.1. Users can explore large multi-gigabyte datasets in seconds, pose exact or approximate similarity queries by drawing data series using the mouse or touch screen and issue random queries on the click of a button. These queries guide the ADS+ index that lies inside RINSE to perform adaptive operations. Users can experience how the index adapts by looking at statistics that are updated on the fly. Additionally, they can compare query answering times, memory footprint, etc. across different access methods, such as a simple scan or the use of a complete (non adaptive) index. Finally, they can also experience the

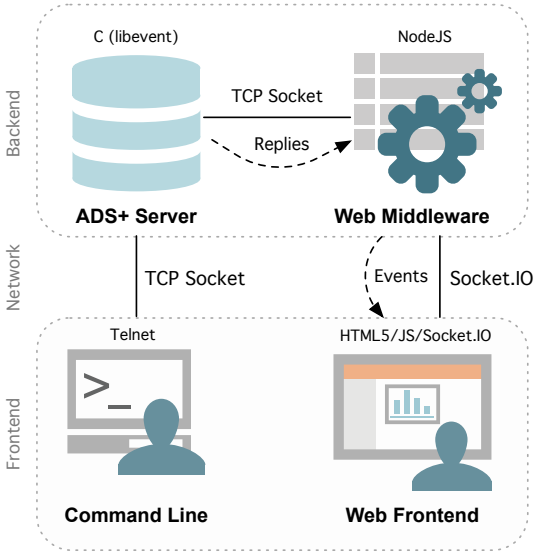



Figure 5.2: RINSE Architecture

differences in terms of data-to-query time by building either a complete or an adaptive index. Users can pose queries using their mouse (or touch screen) or select them from other data collections.

## 5.1 The RINSE System

We now describe RINSE [196], and the use cases that demonstrate the functionality and benefits of adaptive indexing via the ADS+ index.

#### The RINSE System

The overall 3-tiered architecture of the RINSE demonstration system can be seen in Figure 5.2. More information can be found on the ADS+/RINSE website[1].

**Basic Infrastructure.** We developed ADS+ in C and compiled it using GCC 4.6.3 as a shared library (*libads*). We additionally created a TCP server, using *libevent*, to expose its functionality as a network service. This is seen as the ADS+ Server in Figure 5.2. In addition, we created language bindings for *libads* for the NodeJS JavaScript runtime environment. Users connect to the ADS+ Server using a telnet client or a web interface.

The RINSE web interface is developed as a single page application in HTML5, JavaScript and CSS. It connects to a NodeJS middleware using the Socket.IO library.

[1] http://daslab.seas.harvard.edu/rinse/

The middleware has an always active connection to the ADS+ Server. The HTML5 client listens for user events which are pushed to the middleware. The middleware then pushes results back to the client in a real-time event-based mode. This allows for an intuitive and responsive experience where users can draw queries on screen using the mouse (or a touch interface), and see the results appear on screen in near real-time. They can also generate random queries, or choose queries from a list. Data series query workloads [199] can also be used, in order to stress-test ADS+, and demonstrate its performance benefits.

**Supported Features.** RINSE allows users to index data and issue nearest neighbor queries, using three different access methods: 1) a simple serial scan, which reads the complete raw data file for every query; also employing a simple early abandoning technique for avoiding useless computations, 2) a complete iSAX 2.0 [30] index, which is built before the users can start posing queries, and 3) an adaptive ADS+ index, which takes considerably less time to construct. In all three cases, the distance measure used is the Euclidean Distance. Each query can be re-run using a different data structure such that users can see the differences. Furthermore, users have access to statistics measuring the performance of each access method, the amount of data currently ingested by the adaptive index and the memory footprint.

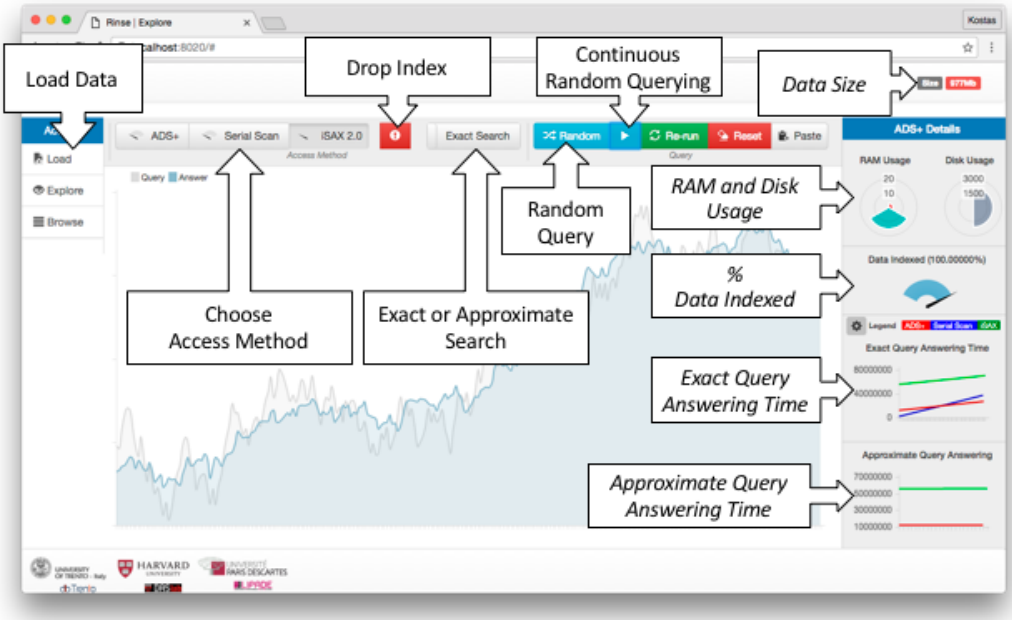



Figure 5.3: RINSE interface components explanation.

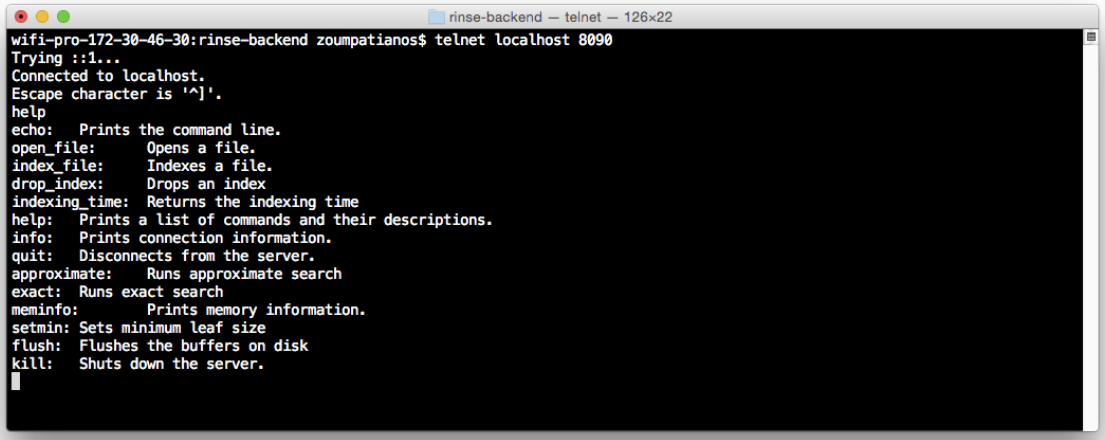



Figure 5.4: An example telnet connection to ADS+.

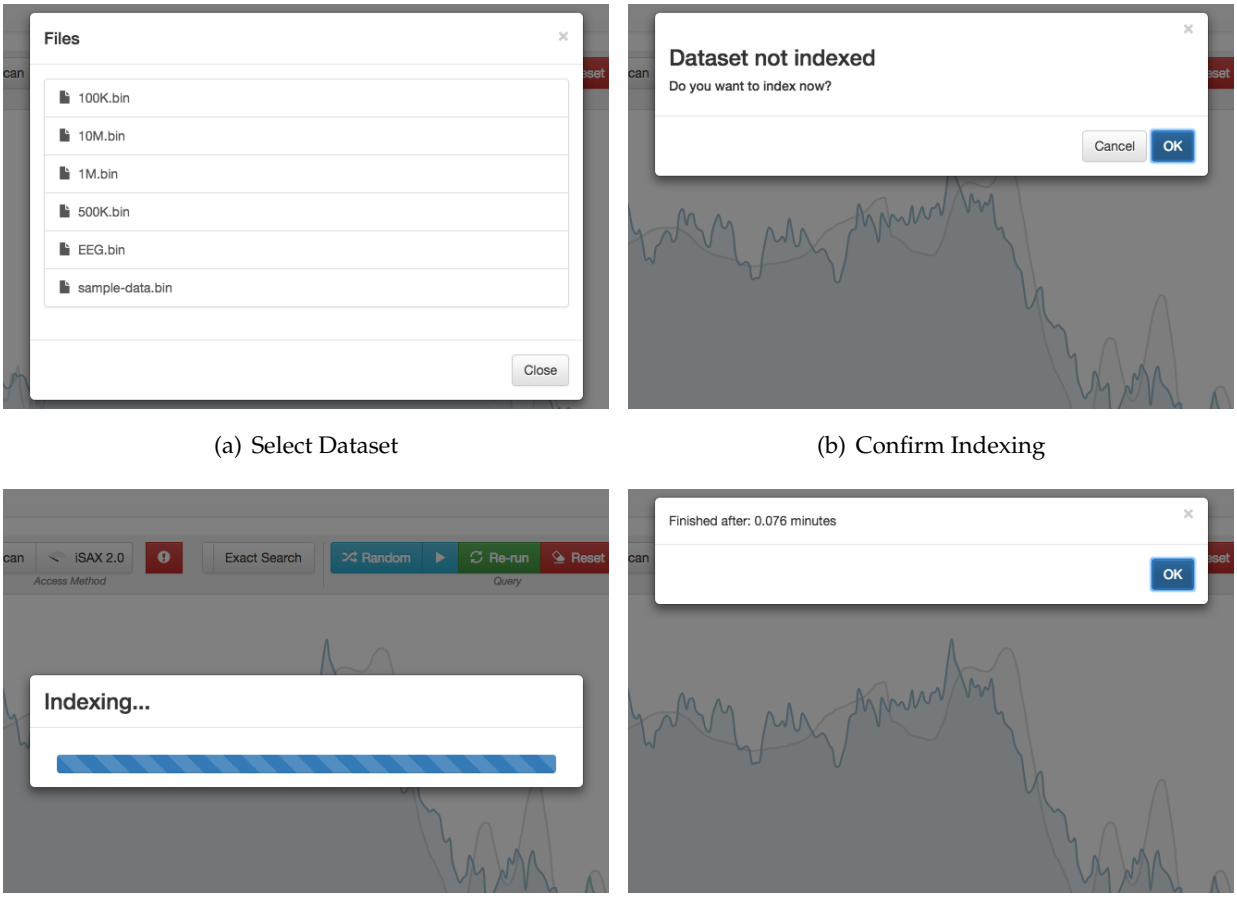
## 5.2 Usage Scenarios

A screenshot of the interface is shown in Figure 5.3. In this figure we also describe the main interface elements and components. There are buttons for issuing random queries, continuous querying and loading data files. Additionally, on the left hand side, there are various information components that describe the current status of each data structure, as well as query answering times. For iSAX 2.0 and ADS+, users are able to run both exact and approximate queries by selecting this option from the RINSE interface. In this way, users can also compare the answering times and accuracy of the various methods when using exact versus approximate processing modes.

Various datasets of different sizes can be easily loaded in the system by the user. The goal of our system is to demonstrate the speed benefits of using partial adaptive indexes over traditional ones for similarity search in large collections of data series, and observe how the system adapts to their queries during data exploration. Bellow we list a set of usage scenarios that showcase the functionality of RINSE.

### 5.2.1 Command Line Connection

All different access methods are socket services listening to different ports. There are three different servers. One for the ADS+ index, one for the full iSAX 2.0 index and one for the serial scan. Users can connect to all these services via telnet. After connecting, users are presented with a command line interface that allows them to query




(a) Select Dataset

(b) Confirm Indexing

(c) Indexing Process

(d) Indexing Finished

Figure 5.5: Dataset Loading and Indexing.

and monitor each data structure. An example of this process can be seen in Figure 5.4, where the user issues the "telnet localhost 8090" command to connect to ADS+. He is then able to interact with the index. For example, as seen in the same figure, issuing a "help" command presents to the user the list of available commands. Those include commands for issuing both approximate and exact queries, for loading datasets, and for getting information about the memory usage of the data structure.

### 5.2.2 Loading and Indexing Data

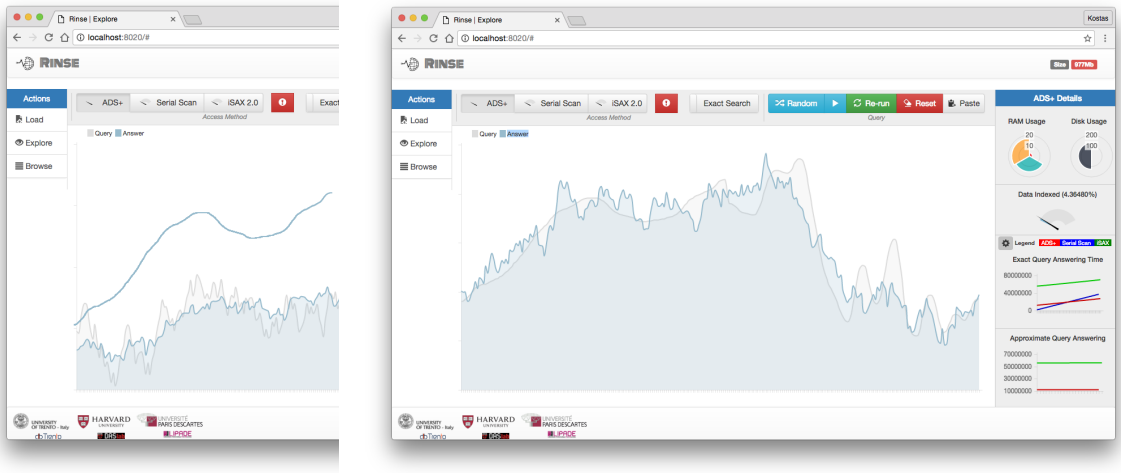
Using the web interface seen in Figure 5.3, users can experience how adaptive indexing allows for quick access to data. We achieve this by directly comparing adaptive indexing to full indexing. The users can initially choose a dataset, by clicking the "Load" button. This process is seen in Figure 5.5. The users can then choose between the three


Figure 5.6: ADS+ After Multiple Queries.

methods and, in case the dataset is not yet indexed, they are presented with a panel that allows them to perform the index construction process. They can then observe how the difference between creating a complete or an adaptive index structure. By repeating the process with increasing data sizes, it becomes evident that adaptive indexing is a scalable approach, while full indexing quickly becomes a bottleneck.

### 5.2.3 Adaptivity Benefits

In this scenario, the users can experience the adaptive nature of ADS+. In particular, we highlight how ADS+ grows incrementally and adaptively as more queries arrive. To achieve this we provide a visual way of monitoring various statistics such as RAM and disk usage. Users can also see the percentage of data indexed by ADS+ at any point in time. In addition, they can observe the index expand as more queries are issued, and also observe the impact that this has on memory usage. Breakdowns for partial and raw data in the index are also provided. All these measures can be seen in Figure 5.6 on the right hand side. As we can see, after multiple queries, the cumulative




(a) Drawing Query (b) Result

Figure 5.7: Drawing Queries with the Mouse.

time to answer all of them is presented in "Exact Query Answering Time" graph of this figure. While ADS+ (the red line), starts with an initial overhead of index construction, it quickly outperforms the serial scan (blue line), which has no initialization cost. For the case of iSAX 2.0 (green line), index construction is a big overhead, which hinders interactivity, even when a big leaf size is used. Moreover, because of our novel exact search algorithm in ADS+ (SIMS), we are still able to outperform the exact query answering times of iSAX 2.0. As a result, iSAX 2.0 will always perform worse than ADS+ and serial scan makes sense only for cases when very few queries have to be answered.

Finally, we provide both a manual and an automatic query process: (a) in the manual case, the index is enriched through queries, which users can draw; (b) in the automatic case, the system automatically executes random queries and enriches the index. To support this scenario, a Play/Pause button is additionally provided to control the automatic query execution. While the system continuously executes random queries, the user can observe the changing index characteristics by following the evolution of the statistics reported graphically, i.e., memory overhead, percentage of data indexed, and query answering times.

### 5.2.4 Data Exploration

In the last usage scenario, the users have the ability to explore large datasets at interactive speeds. The goal of this scenario is to showcase the ability of ADS+ to provide

answers at interactive speeds even though it starts from a partial index state compared to full indexing. Users can issue queries in multiple ways such as interactively drawing data series on screen, as seen in the large plot area on the left hand side of Figure 5.7. For this task one can either use the mouse or a touch screen (e.g., on an iPad). A query is essentially a data series and the system tries to find the closest data series in the database. Additionally, queries can be selected from within existing files of data series; in this scenario the user browses existing files and selects a specific data series to see what is the closest one in the database. The result to each query, i.e., the nearest neighbor, is displayed on screen alongside the query itself as shown in Figure 5.7 on the right. This allows users to visually compare the shapes of the two data series. To demonstrate the benefits of using an adaptive index, users also in this case can compare the query answering times of ADS+ [195] to that of iSAX 2.0 [30], and serial scan.

## 5.3 Summary

In this chapter we presented RINSE, an interactive data exploration platform that demonstrates the benefits of the adaptive data series index ADS+. This system allows the users to experience how adaptive indexing provides quick access to data. It is designed to provide visual access and querying of large data series collections, allowing analysts to quickly browse through their data. Users can additionally use RINSE to perform both full indexing, and simple serial scan, thus comparing the performance of adaptive indexing to these methods. This can happen both in an automatic way, with existing, or random workloads, and in a manual way, where users can draw queries on the screen.

# Chapter 6

# Generating Query Workloads for Data Series Indexes

## 6.1 Introduction

Nearest neighbor queries are of paramount importance, since they form the basis of virtually every data mining and analysis task involving data series. However, such queries become challenging when performed on very large data series collections [31, 134]. The state-of-the-art methods for answering nearest neighbor queries mainly rely on two techniques: *data summarization* and *indexing*. Data series summarization is used to reduce the dimensionality of the data [93, 131, 132, 103, 11, 103, 90, 38, 107] so that they can then be efficiently indexed [131, 165, 30, 180, 13, 156].

We note that despite the considerable amount of work on data series indexing [59, 132, 38, 165, 90], no previous study paid particular attention to the query workloads used for the evaluation of these indexes. Furthermore, since there exist no real data series query workloads, all previous work has used random query workloads (following the same data distribution as the data series collection). In this case though, the experimental evaluation does not take into account the hardness of the queries issued.

Indeed, our experiments demonstrate that in the query workloads used in the past, the vast majority of the queries are easy. Therefore, they lead to results that only reveal the characteristics of the indexes' performance under examination for a rather restricted part of the available spectrum of choices. The intuition is that easy queries are easy for all indexes, and thus these queries cannot capture well the differences among various summarization and indexing methods (the same also holds for extremely hard queries as well). In order to understand how indexes perform for the entire range of

possible queries, we need ways to *measure* and *control* the hardness of the queries in a workload. Being able to generate large amounts of queries of predefined hardness will allow us to stress-test the indexes and measure their relative performance under different conditions.

In this work, we focus on the study of this problem and we propose the first principled method for generating query workloads with controlled characteristics under any situation, without assuming a single summarization and index or a specific test dataset[1]. To this end, we investigate and formalize the notion of *hardness* for a data series query. This notion captures the amount of effort that an index would have to undertake in order to answer a given query, and is based on the properties of the lower bounding function employed by all data series indexes. Moreover, we describe a method for generating queries of controlled hardness, by increasing the density of the data around the query's true answer in a systematic way.

Intuitively adding more data series around a query's nearest neighbor forces an index to fetch more raw data in that area for calculating the actual distances, which makes a query "harder". In this work, we break down this problem into three subproblems.

- Determine how large the area should be around the query's nearest neighbor
- Determine how many data series to add in that area
- Determine how to add data series

The proposed method leads to data series query workloads that effectively and correctly capture the differences among various summarization methods and index structures. In addition, these workloads enable us to study the performance of various indexes as a function of the amount of data that have to be touched. Our study shows that queries of increased hardness (when compared to those contained in the random workloads used in past studies) are better suited for the task of index performance evaluation.

Evidently, a deep understanding of the behavior of data series indexes will enable us to further push the boundaries in this area of research, developing increasingly efficient and effective solutions. We argue that this will only become possible if we can study the performance characteristics of indexes under varying conditions, and especially under those conditions that push the indexes to their limits.

[1]Website: `http://disi.unitn.it/~zoumpatianos/edq`

### 6.1.1 Contributions

- **Index-dependent query answering effort.** We identify the summarization- and query-specific factors that make query answering on data series indexes expensive. Such measures can capture the "effort" a *given index* needs to make in order to answer a *specific query*.
- **Intrinsic query hardness.** We define summarization- and index-independent measures that can effectively capture the "effort" of various summarization/index combinations. We call this intrinsic measure, the "hardness" of a query.
- **Queries with meaningful intrinsic query hardness.** We recommend a set of principles that should be followed when generating evaluation queries such that the intrinsic "hardness" measure can accurately capture various summarization/index-specific query "efforts".
- **Generating workloads.** We describe the first nearest neighbor query workload generator for data series indexes, which is designed to stress-test the indexes at varying levels of difficulty. Its effectiveness is independent of the inner-workings of each index and the characteristics of the test dataset.
- **Experimental evaluation.** We demonstrate how our workload generator can be used to produce query workloads, based on both real and synthetic datasets.

## 6.2 Preliminaries

A data series $\boldsymbol{x} = [x_1, ..., x_d]$ is an ordered list of real values with length $d$. Since data series can be represented as points in a d-dimensional space, in this paper, we also call data series as *points*. Given a dataset $\mathcal{D} = \{\boldsymbol{x}_i\}_1^N$ of $N$ points and a query set $\mathcal{Q} = \{\boldsymbol{q}_i\}_1^M$ of $M$ data series, a query workload $W$ is defined as a tuple $(\mathcal{D}, \mathcal{Q}, k, DIST)$, where each query point $\boldsymbol{q}_i \in \mathcal{Q}$ is a $k$ nearest neighbors ($k$-NN) query and $DIST(\cdot, \cdot)$ is a distance function. When the context is clear, we use $\boldsymbol{x}^{(k)}$ to denote $k$-th nearest neighbor of a query $\boldsymbol{q}$.

In this work, we focus on the nearest neighbor query, i.e., $k = 1$, and define $MINDIST(\boldsymbol{q})$ as $DIST(\boldsymbol{x}^{(1)}, \boldsymbol{q})$. However, our methods can be naturally extended to higher values of $k$ by employing the distance to the $k$-th nearest neighbor. For the rest of this study, we consider the Euclidean distance $DIST(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} - \boldsymbol{y}\|_2$, due to its wide application in the data series domain [30, 165, 180]. Table 6.1 summarizes the notations in this paper.

| Symbol | Description |
|---|---|
| $\boldsymbol{x}$ | A data series |
| $\boldsymbol{q}$ | A query data series |
| $\mathcal{D}$ | A set of $N$ data series |
| $\mathcal{Q}$ | A set of $M$ queries |
| $DIST(\boldsymbol{x}, \boldsymbol{q})$ | Euclidean distance between $\boldsymbol{x}$ and $\boldsymbol{q}$ |
| $MINDIST(\boldsymbol{q})$ | Distance between $\boldsymbol{q}$ and its nearest neighbor |
| $L(\boldsymbol{x}, \boldsymbol{q})$ | Lower bound of $DIST(\boldsymbol{x}, \boldsymbol{q})$ |
| $ATLB(L, \boldsymbol{x}, \boldsymbol{q})$ | Atomic tightness of lower bound of $L$ for $\boldsymbol{x}, \boldsymbol{q}$ |
| $TLB(L)$ | Tightness of lower bound of $L$ |
| $\mu^L(\boldsymbol{q})$ | Minimum effort to answer $\boldsymbol{q}$ using $L$ |
| $\mathcal{N}^\epsilon(\boldsymbol{q})$ | $\epsilon$-Near Neighbors of $\boldsymbol{q}$ |
| $\alpha^\epsilon(\boldsymbol{q})$ | Hardness of $\boldsymbol{q}$ for a given $\epsilon$ |

Table 6.1: Table of symbols.

## 6.3 Characterizing Queries

In this section we investigate the factors that affect the query answering performance of data series indexes and summarizations. We start in Section 6.3.1 with an introduction on lower bounding functions used by summarizations, and study the *minimum effort* that an index can make in order to answer a given query. We continue our discussion with the requirements that should be satisfied for defining an index-dependent measure of query answering effort. In order to do this, we connect the de facto measure for quantifying the quality of a summarization, called the tightness of the lower bound (TLB), to the percentage of data that an index will need to check, in the best case, in order to answer a query.

We note that there exists an implicit relationship between summarizations and indexes: each node of an index corresponds to a summarization of all the data series below it. As a result, data series indexes can be thought of as hierarchical (multi-level) summarizations. This is illustrated in Figure 2.5, where each internal node of the index includes a summary of all the data series below it.

**Requirements for a reasonable hardness definition.** Our goal in this work is twofold. First, to be able to generate queries with meaningful effort values, effectively capturing the quality of the summarizations, and second, to have a meaningful intrinsic hardness definition that effectively captures these efforts. For this reason, we need queries and a hardness definition with the following properties:

1. **Inter-index (intra-query) effort accuracy.** Given *a single query* $q_1$ and *two indexes* $I_1$ and $I_2$, since data-series indexes are multi-level summarizations of data series,

we can safely consider that at each level $l_{I_1}$ of index $I_1$, this index summarizes data series with a specific summarization $Summarization(l_{I_1})$. The summarization error is a commonly accepted measure of how good a summarization is, and at each level $l_{I_1}$ of index $I_1$, it is defined as SummError$(l_{I_1})$, and SummError$(l_{I_2})$ for index $I_2$. Given two summarization errors, for a specific level of each index (or two different levels of the same index in a general case), the corresponding efforts should capture how much bigger the error of one level is over the other level. Formally, if the effort values for answering the query using each summary are Effort$(l_{I_1}, q_1)$, Effort$(l_{I_2}, q_1)$, it should hold, for every level of each index that:

$$\frac{\text{Effort}(l_{I_1}, q_1)}{\text{Effort}(l_{I_2}, q_1)} = \frac{\text{SummError}(l_{I_1})}{\text{SummError}(l_{I_2})}$$

This property would ensure that the worse a summarization the more data it has to check. Additionally, it would ensure that there are no cases where the distribution of the data is such that bad summarizations and good summarizations have the same performance. It is important to note that this is a property of the queries, and not a property of our hardness measure. We should either generate, or select queries that respect this property.

2. **Intra-index (inter-query) hardness accuracy.** If for *two queries* $q_1, q_2$ and *a single index* $I_1$, the effort of answering these queries using this index, at every level $l_{I_1}$ is Effort$(l_{I_1}, q_1)$, Effort$(l_{I_1}, q_2)$, an ideal intrinsic hardness definition would need to accurately capture how much bigger the effort for $q_2$ is over the effort of $q_1$ *for this index level*. Formally, if the hardness values for $q_1$ and $q_2$ are Hardness$(q_1)$ and Hardness$(q_2)$, it should hold that:

$$\frac{\text{Effort}(l_{I_1}, q_1)}{\text{Effort}(l_{I_1}, q_2)} = \frac{\text{Hardness}(q_1)}{\text{Hardness}(q_2)}$$

We argue that a hardness definition that satisfies these criteria across different summarizations/indexes and different queries is an ideal intrinsic hardness definition, which will subsequently allow us to generate queries using a predefined hardness, effectively causing the appropriate relative efforts to various different indexes/summarizations.

Since, as we already mentioned, we consider data series indexes as multi-level summarizations of data series, in the rest of the section we reason using summarizations. We assume that a bad summarization, which does not allow a lot of pruning to be performed, corresponds to a very rough representation of the indexed data, commonly

found in higher level nodes of an index hierarchy, where it summarizes a large quantity of data series. On the contrary, a good summarization corresponds to a very precise representation of the data series, it allows more pruning to be performed, and is commonly found at the leaf level of an index, where it summarizes a smaller quantity of the data; usually the size of a disk-page. We consider that the better the pruning at each level, and the smaller the amount of levels, the better the performance of the index in overall will be, and as a result we can concentrate our study on summarizations of varying precision as a proxy to quantifying global index effort and query hardness.

### 6.3.1 Index-dependent query answering effort

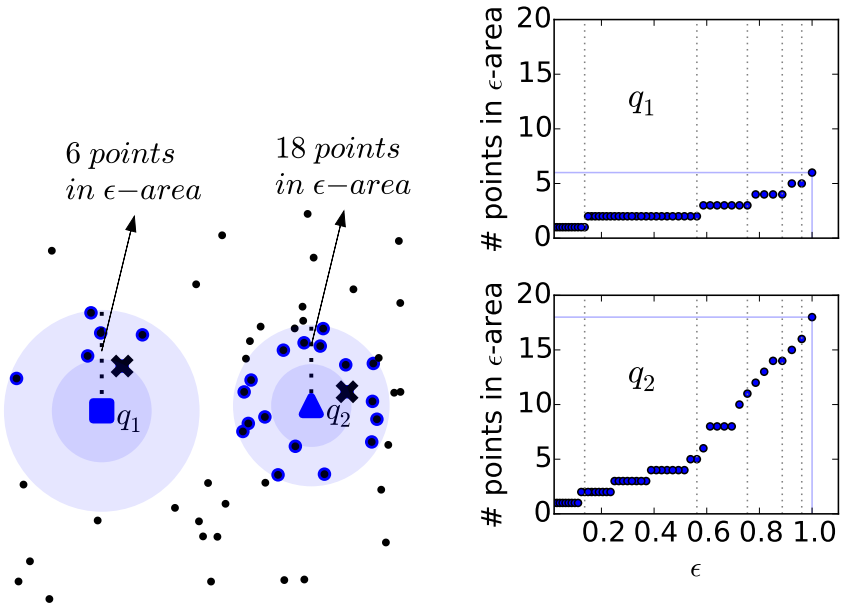
#### Lower Bounds, ATLB and TLB

When navigating an index, we make use of the lower bounds (computed based on the summarizations) of the true distances of data series in the original space. This technique guarantees that there will be no false negatives in the candidate set, but it does not exclude the false positives. Therefore, the indexes will either have to move to lower (and more precisely summarized) levels of the index that contain better summarizations, performing further lower bound computations, or in the case of a leaf node, when the summarizations cannot be further improved, they need to fetch the raw data series as well, and check them before returning the answer, filtering out the false positives, and thus guaranteeing the correctness of the final answer.

The use of lower bounds can be conceptually thought of as the cut-off point in the distance between two summarized data series. Below this point, the corresponding raw data, or lower and more precisely summarized levels of the index have to be checked. To capture this notion, we can use the Tightness of Lower Bound ($TLB$) [179], which is measured as the average ratio of the lower bound over the true distance. We formalize this notion by first introducing here the Atomic Tightness of Lower Bound ($ATLB$), which is the same ratio, but defined for a specific query and summarized data series pair.

**Definition 8 (Atomic Tightness of Lower Bound)** *Given a summarization with lower bounding function $L$, the atomic tightness of lower bound ($ATLB$) between a data series $\boldsymbol{q}$ and a summary of data series $\boldsymbol{x}$ is defined as*

$$ATLB(L, \boldsymbol{x}, \boldsymbol{q}) = L(\boldsymbol{x}, \boldsymbol{q}) / DIST(\boldsymbol{x}, \boldsymbol{q}) \tag{6.1}$$




(a) Points in $\epsilon$-area ($\epsilon = 1.0$) (b) # of points in $\epsilon$-area

Figure 6.1: Two random queries with nearest neighbors depicted with "×".

**Definition 9 (Tightness of Lower Bound)** *Given a summarization with lower bounding function $L$, a set of queries $\mathcal{Q}$ and a set of data series $\mathcal{D}$, the tightness of lower bound ($TLB$) for this summarization is defined as*

$$TLB(L) = \frac{1}{N \times M} \sum_{\boldsymbol{q} \in \mathcal{Q}} \sum_{\boldsymbol{x} \in \mathcal{D}} ATLB(L, \boldsymbol{x}, \boldsymbol{q}) \tag{6.2}$$

**Example 1** *Figure 6.1(a) demonstrates the implications of the $TLB$ . For simplicity, we represent each data series as a* point *in a two-dimensional space, i.e., $d = 2$. In this example, we plot two queries $\boldsymbol{q}_1, \boldsymbol{q}_2$ and mark their nearest neighbors with a bold "×". Assume $MINDIST(\boldsymbol{q}_1) = 0.33$, $MINDIST(\boldsymbol{q}_2) = 0.26$, and all data series are summarized using the same summarization method. Let the $ATLB$ between the queries and any data point be 0.5, i.e., the lower bound of the distances between $\boldsymbol{q}_1$ or $\boldsymbol{q}_2$ and all other points is 0.5 times their actual distance. According to the definition of $ATLB$, a point $\boldsymbol{x}$ cannot be pruned if*

$$L(\boldsymbol{x}, \boldsymbol{q}) \leq MINDIST(\boldsymbol{q}) \tag{6.3}$$

$$\Leftrightarrow DIST(\boldsymbol{x}, \boldsymbol{q}) \leq \frac{MINDIST(\boldsymbol{q})}{ATLB(L, \boldsymbol{x}, \boldsymbol{q})}. \tag{6.4}$$

*This means that for $\boldsymbol{q}_1$, all points whose actual distance is within a radius $\rho = \frac{0.33}{0.5}$ from $\boldsymbol{q}_1$'s nearest neighbor can not be pruned, because their lower bound distances are less than the*

*distance to the answer. Since* $ATLB(L, \boldsymbol{x}, \boldsymbol{q}) \in (0, 1]$*, the right hand side of Inequality* (6.4) *is always no less than* $MINDIST(\boldsymbol{q})$*. These ranges are depicted as disks in Figure 6.1(a).*

Note that the *TLB* is small for an inaccurate summarization, i.e. such summarization tends to significantly underestimate the distance. As a result, data series under this summarization will look much closer than they actually are. Consequently, the index will have to check more raw data, leading to a longer execution time. This is indeed an important criteria for measuring index performance. To formalize such difference in the effort of more or less effective summarization/indexing techniques, we hereafter introduce the notion of *Minimum Effort*. We will then examine in the following subsections how this notion can be linked to a meaningful concept of hardness of a query within a workload.

#### Index-dependent measure of Minimum Effort

We define Minimum Effort (*ME*) as the ratio of points over the total number of series that an index has to fetch to answer a query, given that it uses the best summarization that it contains for each distinct data series. This aims to describe the absolute minimum amount of raw data that an index will have to touch in order to answer a query.

**Definition 10 (Minimum Effort)** *Given a query* $\boldsymbol{q}$*, its* $MINDIST(\boldsymbol{q})$ *and a lower bounding function L, the minimum effort that an index using this lower bounding function has to do in order to answer the query is defined as*

$$\mu^L(\boldsymbol{q}) = |\{\boldsymbol{x} \in \mathcal{D} | L(\boldsymbol{x}, \boldsymbol{q}) \leq MINDIST(\boldsymbol{q})\}| / |\mathcal{D}|$$

As we have seen in Example 1, given a fixed *ATLB* between the query and data series, data series that contribute to *ME* are within a radius $\rho = \frac{MINDIST(\boldsymbol{q})}{ATLB(L, \boldsymbol{x}, \boldsymbol{q})}$ from the query's nearest neighbor and these data series cannot be pruned. The size of this radius is inversely proportional to $ATLB$ and proportional to $MINDIST(\boldsymbol{q})$.

It is important to clarify that this is the *minimum* possible effort that an index will have to undertake, and in most cases it will be smaller than the actual effort that the index will actually do. This is because the search for the solution hardly ever starts with the real answer as a best-so-far. Additionally, the more lower bounds are computed, the slower the index becomes.

To demonstrate this we have run a small experiment with 100,000 synthetic data series, generated with a simple randomwalk, a process which we will describe later on

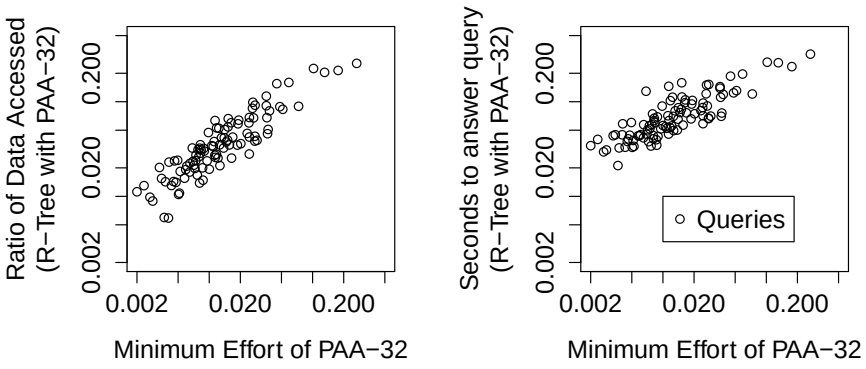



Figure 6.2: 100,000 random walk generated data series and 100 queries.

in the text, and 100 queries. We converted all data series into PAA with 32 segments and built an R-Tree index above them. We then fired the 100 queries to the index and measured the query answering time as well as the ratio of the data accessed in order to answer each query. Further on, for every query we measured its minimum effort as per our definition and we present the results in Figure 6.2. We can see that the minimum effort is highly correlated to both the query answering time and the amount of data that an index checks. Additionally, we can see that the grouping of data by the index causes an additional overhead, since the ratio of data checked by the R-Tree is constantly higher than the minimum effort and the time is higher than simply accessing the data, because of the costly index traversal.

### 6.3.2 Intrinsic query hardness

#### $\mathcal{E}$-dependent hardness

Recall that our goal is to investigate the general, summarization-independent, intrinsic hardness of queries in a workload. Since Minimum Effort is tied to a specific summarization, we need a more general notion to capture how hard a query is. Intuitively, the hardness of a query is related to the number of points around its nearest neighbor (true answer). Given this intuition, we define the $\epsilon$-Near Neighbors ($\epsilon$-NN) of a query $\boldsymbol{q}$ as follows.

**Definition 11 ($\epsilon$-Near Neighbors)** *Given $\epsilon \geq 0$, the $\epsilon$-near neighbors of a query $\boldsymbol{q}$ is $\mathcal{N}^{\epsilon}(\boldsymbol{q}) = \{\boldsymbol{x} \in \mathcal{D} | DIST(\boldsymbol{x}, \boldsymbol{q})$*
*$\leq (1+\epsilon) MINDIST(\boldsymbol{q})\}$, i.e., all the points in $\mathcal{D}$ that are within $(1+\epsilon) MINDIST(\boldsymbol{q})$ of*

*the query's nearest neighbor.*

The $\epsilon$-NN naturally defines a hypersphere around the nearest neighbor of the query. In the rest of this paper, we will refer to this hypersphere as the **$\epsilon$-area**. Now we define the $\epsilon$-hardness of a query as follows.[2]

**Definition 12 ($\mathcal{E}$-hardness)** *Given $\epsilon \geq 0$, the $\epsilon$-hardness of a query $\boldsymbol{q}$ is $\alpha^{\epsilon}(\boldsymbol{q}) = \frac{|\mathcal{N}^{\epsilon}(\boldsymbol{q})|}{|\mathcal{D}|}$, i.e., the fraction of $\mathcal{D}$ that is within $(1+\epsilon)MINDIST(\boldsymbol{q})$ of the query.*

**Example 2** *Going back to the example in Figure 6.1(a) let us assume that the total number of points in the dataset is 100. $\mathcal{E}$-hardness computation for $\epsilon = 1.0$ accordingly yields $\alpha^{1.0}(\boldsymbol{q}_1) = 0.06$ and $\alpha^{1.0}(\boldsymbol{q}_2) = 0.18$.*

In the following, we will focus on showing under which conditions the $\epsilon$-hardness can be used as a meaningful intrinsic hardness measure for queries within a workload.

#### $\mathcal{E}$-independent hardness

As sketched in the introduction, we are interested in an intrinsic hardness measure for queries that would essentially enable a meaningful comparative analysis of the performances of various summarization/indexing techniques.

We have introduced above the notion of Minimum Effort, which provides a good indication of the efficiency of a summarization when dealing with a query. What we would like now, are the following two objectives:

1. To select or generate queries (irrespective of their hardness values), which are able to accurately *differentiate*, or discriminate, between more and less efficient summarizations. This is the inter-index (intra-query) effort accuracy property that we talked about in the beginning of Section 6.3.

2. Define an $\epsilon$-independent intrinsic hardness measure, whose values, corresponding to the queries in a workload are *representative of the Minimum Effort*, and this moreover for *all the summarizations under test*. This is the intra-index (inter-query) intrinsic hardness accuracy property.

We will focus hereafter on detailing the criteria needed for $\epsilon$-hardness to hold as a robust hardness measure for the queries within a workload, according to the requirements stated above.

[2] A similar definition can also be found in [21].

**The choice of an $\epsilon$ value.** Let us go back to Examples 1 and 2 and assume as previously a summarization with a (constant) $ATLB$ of 0.5. As we have shown above, points within a radius of $\frac{0.33}{0.5} = 0.66$ from $q_1$ all participate in the Minimum Effort of our considered summarization. This radius, on the other hand, corresponds to an $\epsilon$-area for $\epsilon$=1. For this precise value of $\epsilon$, we are indeed sure that the respective $\epsilon$-area covers exactly the points participating in the ME. It turns out that, following directly from Inequality (6.4), to ensure that an $\epsilon$-area around a query $q$ in a workload covers all the points that participate in the Minimum Effort of a specific summarization/index *with a constant ATLB*, the following must hold:

$$\epsilon \geq \frac{1}{ATLB(L, \boldsymbol{x}, \boldsymbol{q})} - 1 \tag{6.5}$$

When enforcing an equality in the above formula, one obtains an $\epsilon$ value for which the corresponding area contains *all and precisely* the points involved in the Minimum Effort of the specific summarization. Accordingly, the $\epsilon$-hardness is equal to the Minimum Effort.

Let us assume now that for our example we had picked an $\epsilon = 0.1$ for the hardness computation. The computed hardness would then be *expectedly lower* that the Minimum Effort of our considered summarization, since all ME points between $\epsilon = 0.01$ and $\epsilon = 1.0$ will be "unaccounted for". Assume further that in the interval between $\epsilon = 0.01$ and $\epsilon = 1.0$ we would have a very large number of dataset points. The estimated hardness of $q_1$, computed as $\epsilon$-hardness for $\epsilon = 0.01$, would then be *significantly lower* than the actual Minimum Effort. The particular difference would be furthermore uncontrolled by the query hardness.

By the above, we argue that in order to be able to ensure that $\epsilon$-hardness is a meaningful hardness measure within a workload, the value of $\epsilon$ to be chosen should be such that few/no points participating in the Minimum Effort of the summarizations tested are located outside the $\epsilon$-areas. Considering again the case of constant $ATLB$s, it is obvious that if this "coverage" property holds for a summarization it will hold for "better" summarizations (i.e. with higher $ATLB$) as well. In fact, ensuring an equality in equation 6.5 for the *worst* summarization tested guarantees that the ME points are covered by $\epsilon$-areas for all summarizations.

Let us now go back to our original question: *which $\epsilon$ value should one choose* to ensure a meaningful hardness measure? The answer to this question can be seen as simply: *Any value*, but the lower such value the lesser the range of summarizations that can in principle be meaningfully tested by employing the respective workload. This $\epsilon$ value,

should be chosen to correspond to the worst summarization that we are interested in penalizing. Below this point all equally bad or worse summarizations will be facing the same effort. As a result we need an $\epsilon$ value large enough to cover all reasonably bad indexes, but not large enough for penalizing outlying bad summarizations as they are already too slow.

**Structure of the $\epsilon$-area.** The choice of a "convenient" $\epsilon$ value ensuring that no ME points are "left aside" is a first necessary step towards turning $\epsilon$-hardness into an intrinsic hardness measure. However, as we will hereafter show, this first step is not sufficient.

**The intra-index (inter-query) hardness accuracy property.** As we mentioned in the second requirement of the beginning of Section 6.3, we need to ensure that the effort of answering queries using a single summarization is accurately captured by our hardness definition. That is, for any given index/summarization, the ratio of the efforts for this *single summarization* and *two queries* should be equal to the ratio of the hardnesses of those queries.

Going back to our example, we had there two queries $q_1$ and $q_2$ with respective hardnesses (computed as $\epsilon$-hardnesses for $\epsilon = 1.0$) of 0.06 and 0.18 respectively. These values suggest that $\boldsymbol{q}_2$ is 3 times harder than $\boldsymbol{q}_1$. This indeed, as shown, holds for our example summarization technique with $ATLB = 0.5$, whose Minimum Effort is indeed characterized by the $\epsilon$-hardness for $\epsilon = 1.0$. We will hereafter denote this summarization by $S_1$.

It is important to note that this property holds when a specific and common summarization is used to answer both queries, and is easy to satisfy this requirement. Simply, since $\epsilon$ is fixed, each area should have a specific amount of times more points than the other. Let us then look at a "better" summarization with a (constant) $ATLB = 0.83$, denoted hereafter by $S_2$. It is easy to show that $S_2$'s Minimum Effort on $q_1$ is 0.02. However, $S_2$'s ME on $q_2$ is also 0.02! Accordingly, for $S_2$, the two queries then appear as *equally hard*.

The question that arises then is: how could one ensure $q_1$ is 3 times harder than $q_2$ for $S_2$, and in general for any other summarization within the limits of those addressed by the chosen $\epsilon$? In other words, how can we make this property hold for all possible summarizations. It turns out that, according to equation 6.5, $S_1$'s Minimum Effort is best characterized by the $\epsilon$-hardness computed for $\epsilon = 0.2$, which in fact yields $\alpha^{0.2}(\boldsymbol{q}_1) = 0.02$ and $\alpha^{0.2}(\boldsymbol{q}_2) = 0.02$. What we would want instead is for the 1:3 ratio to still hold at $\epsilon = 0.2$ and for any other $\epsilon$. As a result, for all summarizations we would

like the following to hold:

$$\frac{\alpha^{\epsilon}(\boldsymbol{q}_1)}{\alpha^{\epsilon}(\boldsymbol{q}_2)} = \frac{\alpha^{\hat{\epsilon}}(\boldsymbol{q}_1)}{\alpha^{\hat{\epsilon}}(\boldsymbol{q}_2)}, \forall \hat{\epsilon} \leq \epsilon \tag{6.6}$$

Note that the lack of this property for $\epsilon = 1.0$ on our example's query workload comprising $q_1$ and $q_2$ *makes this workload unsuitable for comparative analysis of* $S_1$ *and* $S_2$, and in fact for any summarization set comprising $S_2$ and several other "better" summarizations. The issue here is indeed related to the *structure* of the $\epsilon$-areas around $q_1$ and $q_2$, i.e. how points are distributed in these areas. To correct such issue, one needs to either *filter-out* unsatisfying queries or *alter* their structure. We will show in Section 6.5 the operational means for such structure change.

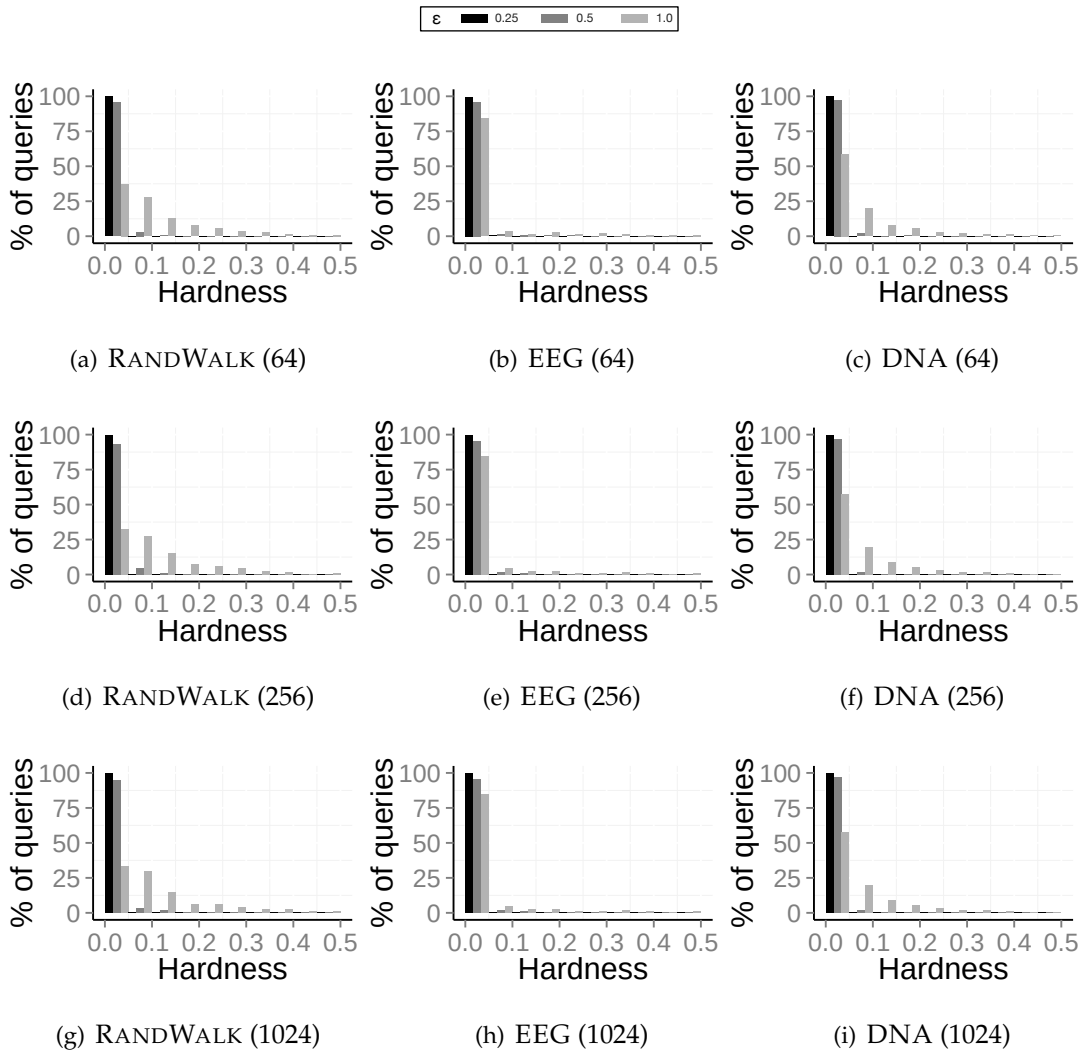
**The inter-index (intra-query) effort accuracy problem.** Recall moreover that, we need to also ensure that the first requirement defined in the beginning of Section 6.3 holds. We need to ensure the *discriminative* power of each query. This means that, within a single query, if $\epsilon$-hardness does not increase proportionally to the summarization errors and their corresponding $\epsilon$ values, this query could turn out as *unfit for differentiating the quality of the tested summarizations*. The query could indeed appear as equally hard for summarizations whose quality is objectively very different, or it could appear disproportionately hard for some summarizations and disproportionately easy for others. Intuitively, the difference in effort across different summarizations would not be representative of the actual quality of the summarization, but instead it would be artificially affected by a biased placement of points around the query. We will call this issue the *inter-index (intra-query) effort accuracy problem*.

As we've already mentioned in the beginning of the section, to avoid this problem, for *any single query* that we choose or generate, it should hold that the effort for a specific summarization over the effort of any other one should be proportional to their relative summarization errors.

Each summarization has a specific summarization error which can be meaningfully measured as: $1 - TLB$. The smaller this number, the smaller the error (the better the TLB), and the larger this number, the bigger the error (the worst the TLB). We would like the following to hold for any query $q_1$ and any two different summarizations $S_1$ and $S_2$:

$$\frac{\mu^{S_1}(\boldsymbol{q}_1)}{\mu^{S_2}(\boldsymbol{q}_1)} = \frac{1 - TLB(S_1)}{1 - TLB(S_2)} \tag{6.7}$$

Note that the criteria derived above are once again a structural condition that the $\epsilon$-areas in a workload need to respect for the workload to be valid.




(a) RANDWALK (64) (b) EEG (64) (c) DNA (64)

(d) RANDWALK (256) (e) EEG (256) (f) DNA (256)

(g) RANDWALK (1024) (h) EEG (1024) (i) DNA (1024)

Figure 6.3: Histograms of query hardnesses.

In summary, an $\epsilon$-independent hardness definition requires that the properties expressed in Equations (6.6) and (6.7) hold. In Section 6.5, we describe a method for generating queries and injecting points in existing datasets in such a way that these properties hold.

## 6.4 Evaluation of previous work

### 6.4.1 Datasets and Workloads

In this section, we review some common datasets and their corresponding workloads that have been used in the literature. We use the same datasets in our experimental

section as well. For capturing trends and shapes, we z-normalize (mean=0, std=1) all data series following common practice.

RANDWALK [11, 59, 131, 38, 165, 13, 30, 90, 99, 31, 156, 195]. This dataset is synthetically produced using a random walk data series generator. The step size in each data series varies according to a Gaussian distribution. We start with a fixed random seed and produce 200,000 data series of length 1024, 256 and 64.

EEG [189, 91, 13, 99, 134]. We use the electroencephalograms dataset from the UCI repository [17], and sample 200,000 data series of length 1024, 256 and 64 from the dataset to be used as the dataset.

DNA [30, 31, 195]. We use the complete Human DNA (Homo Sapiens), obtained from the Ensembl project.[3] We sample the dataset to create 200,000 data series of length 1024, 256 and 64.

The query workloads that have been used in all past studies are generated in one of the following two ways.

1. A subset of the dataset is used for indexing and a disjoint subset is used as queries [30, 90, 31, 195].

2. The entire dataset is used for indexing. A subset of it is selected and a small amount of random noise is added on top. This is used as the query set [11, 103, 99].

In our study, we shuffle the datasets, and use half of each dataset (100,000 data series) as queries, and the other half (100,000 data series) as the indexed dataset.

### 6.4.2 Hardness Evaluation

One of our key requirements is the ability to test how indexes scale as they need to check an increasing amount of data. This is the case with hard queries, for which indexes are not able to easily identify the true nearest neighbor. In this subsection, we choose 1,000 random queries from our initial query set and evaluate the hardness of each one of them for $\epsilon \in \{0.25, 0.5, 1\}$.

The results are depicted in Figure 6.3. As the histograms show, for all data series (length 64, 256, 1024) the query workloads are mainly concentrated towards easy queries. For $\epsilon = 0.5$, the average hardness is less than 0.1, while for $\epsilon = 1.0$, the average of hardness is less than 0.25. Additionally, as the $\epsilon$ decreases to 0, the hardness of the queries drops very rapidly for both the RANDWALK and the DNA datasets. These

[3] ftp://ftp.ensembl.org/pub/release-42/

low hardness values further motivate us for the need of a controlled way to generate workloads with queries of varying hardness.

## 6.5 Query workload generation

As we demonstrated in the previous section, all the widely-used (i.e., randomly generated) query workloads are biased towards easy queries. In this paper, we argue that an effective query workload should contain queries of varying hardness. Since most existing queries are easy, we start with those easy queries and make them harder by adding more points in their $\epsilon$-areas, i.e. by a process of *densification*.

We start with a list of different hardness values in non-decreasing order $[\alpha_1^\epsilon, ..., \alpha_n^\epsilon]$ with respect to some $\epsilon$ that is provided by the user ($\sum_{i=1}^{n} \alpha_i^\epsilon \leq 1$, $\alpha_i^\epsilon \leq \alpha_j^\epsilon$ for $i < j$), and an input sample query set $\mathcal{Q}$ that contains many easy queries (produced through random generation).

We will split our workload generation in three stages:

- First, we will select a subset $\mathcal{Q}'$ of $\mathcal{Q}$ comprising queries whose $\epsilon$-areas do not intersect. This will ensure that the densification process can be applied individually to each of the selected queries, without side-effects on the rest of $\mathcal{Q}'$. Indeed, figure 6.4 shows an example that $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ intersect, making individual hardness difficult to control.
- Next, we will match (a subset of $n$ chosen) queries in $\mathcal{Q}'$ with the provided hardness values and identify the amount of points we need to add in each $\epsilon$-area.
- Finally, we will spread these points in such a way that as the $TLB$ of the index gets worse, the minimum effort captured by the workload increases, following our intuitions described in Section 6.3 and Example 1.

The following subsections describe in more details each of the three stages listed above.

### 6.5.1 Generating Non-intersecting Queries

Recall that the first stage in our query workload generation is that of selecting, within the initial set of provided queries $\mathcal{Q}$, a subset $\mathcal{Q}'$ thereof comprising queries whose pairwise $\epsilon$-areas do not intersect. Clearly, we also wish that $\mathcal{Q}'$'s cardinality be maximized, so as to accommodate a wide range of possible hardness values. We show hereafter two approaches towards achieving our query selection goal.

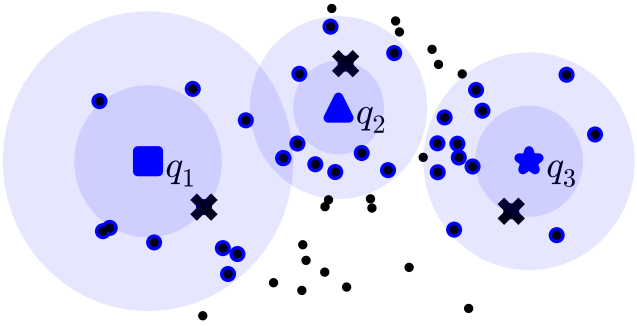



Figure 6.4: Example of 3 queries, where the $\epsilon$-area of $q_1$ and $q_2$ intersect. As a result we cannot control the hardness of these two queries independently, as densifying each one of the two zones might affect also the hardness of the other query.

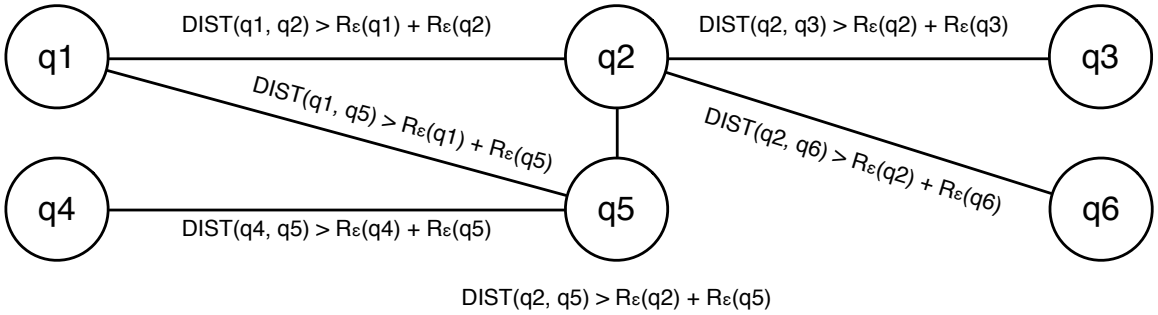



Figure 6.5: Maximal clique formed by $q_1$, $q_2$, and $q_5$.

In our first approach of query selection, also presented in [199], we aim at reasoning on $\epsilon$-areas built by considering existing nearest neighbors, without altering them.

Our first step is to calculate the radius of each $\epsilon$-area. In order to do this we need to find the distance to the nearest neighbor and multiply it by $(1+\epsilon)$. Since we are using Euclidean distance (a metric), we can use the triangle inequality in order to find non-intersecting queries.

Given a distance function $DIST$ in metric space, we set $R_\epsilon(\boldsymbol{q}) = (1+\epsilon)MINDIST(\boldsymbol{q})$ as the radius of the $\epsilon$-area. Two queries $\boldsymbol{q_i}, \boldsymbol{q_j} \in \mathcal{Q}, \boldsymbol{q_i} \neq \boldsymbol{q_j}$ are non-intersecting if and only if the following holds (by the triangle inequality):

$$DIST(\boldsymbol{q}_i, \boldsymbol{q}_j) > R_\epsilon(\boldsymbol{q}_i) + R_\epsilon(\boldsymbol{q}_j)$$

In order to validate this constraint, we first need to calculate all the pairwise distances for all the queries in $\mathcal{Q}$, and for each query $\boldsymbol{q}$, the distance to its nearest neighbor

**Algorithm 11:** FindNonIntersectingQueries

```
1  R_ε = createRadius(Q, D, ε);
2  g = createVerticesFromQueries(Q);
3  for (q_i, q_j) ∈ Q × Q do
4      if DIST(q_i, q_j) > R_ε(q_i) + R_ε(q_j) then
5          g.addEdge(q_i, q_j);
6  V = g.getSortedVertices(); // Sorted by ascending degree
7  Q' = ∅;
8  for q ∈ V do
9      if isCompatible(q, Q') then
10         Q' = Q' ∪ {q};
11 return Q';
```

$MINDIST(\boldsymbol{q})$.

Given the set of queries and their pairwise distances, we can create a graph $\mathcal{G}$, where each vertex represents a query, and an edge exists if two queries $\boldsymbol{q}_i$ and $\boldsymbol{q}_j$ do not interfere with each other. Now it is clear that our problem is closely related to the maximum clique problem. Figure 6.5 illustrates an example graph with 6 queries, where queries $\boldsymbol{q}_1$, $\boldsymbol{q}_2$, $\boldsymbol{q}_5$ form the maximum clique, being mutually non-intersecting.

Note that finding the maximum clique in graph $\mathcal{G}$ is NP-complete, we therefore employ a greedy approach to select queries by assigning a query $\boldsymbol{q}$ to some $\alpha_i^\epsilon$ (denoted as $\boldsymbol{q}(\alpha_i^\epsilon)$) if its current hardness is smaller than $\alpha_i^\epsilon$ and if its $\epsilon$-area does not intersect with the $\epsilon$-areas of all previously assigned queries. This ensures that when densifying the $\epsilon$-area for $\boldsymbol{q}(\alpha_i^\epsilon)$, the hardness of other selected queries $\boldsymbol{q}(\alpha_j^\epsilon)$ ($j \neq i$) will remain unaffected.

Algorithm 11 describes how to find non-intersecting queries. The algorithm sorts the vertices of the graph based on their degree. The intuition is that high-degree vertices have more compatible vertices. We then keep reading vertices in that order, adding compatible ones to a list while skipping incompatible ones.

### 6.5.2 Generating Non-intersecting Queries with Synthetic Nearest Neighbors

Depending on the specific data and query set, as well as on the quantity of hardness values provided, the strategy presented in the previous subsection may lead to insufficient selected queries. We hereafter focus on a new approach, which consists in *synthetically generating nearest neighbors* for some (possibly all) of the queries in the initial

set $\mathcal{Q}$. The main advantage of this procedure is that upon exiting the synthetic nearest neighbors generation, $\mathcal{Q}' = \mathcal{Q}$, that is, all provided queries can be part of our workload.

**Synthetic Nearest Neighbor generation**

We will start by computing the minimum distances from each query in $\mathcal{Q}$ to each of the other queries in $\mathcal{Q}$. For a given query $\boldsymbol{q}$ we will denote this distance by $MINDISTQ(\boldsymbol{q})$. We will then set:

$$R_\epsilon(\boldsymbol{q}) = \frac{MINDISTQ(\boldsymbol{q})}{2} - \omega$$

where $\omega$ is a very small quantity necessary to avoid $\epsilon$-areas tangency.

The above defines a valid radius for placing $\epsilon$-spheres around each query in $Q$. Based on this computed radius, we then define for each of the queries in $Q$ a synthetic nearest-neighbor distance:

$$MINDIST_{syn}(\boldsymbol{q}) = \frac{R_\epsilon(\boldsymbol{q})}{1+\epsilon}$$

Finally, for each query $\boldsymbol{q}$, we either keep its existing (i.e. original dataset) nearest neighbor or generate a new, synthetic one, according to the following:

- if $MINDIST(\boldsymbol{q}) \leq MINDIST_{syn}(\boldsymbol{q})$ then the nearest neighbor stays unchanged.
- else, we set $p_q$ to be a new point, uniformly generated with respect to the constraint $DIST(\boldsymbol{q}, p_q) = MINDIST_{syn}(\boldsymbol{q})$ and we let $\mathcal{D} = \mathcal{D} \cup p_q$.

**Generating unconstrained normalized points**. A (z-) normalized N-dimensional point $\boldsymbol{x} = [x_1, \ldots, x_N]^T$ has mean 0 and standard deviation 1, that is, it respects the following two equations:

$$\sum_{i=1}^{N} x_i = 0 \tag{6.8}$$

$$\sum_{i=1}^{N} x_i^2 = N \tag{6.9}$$

Note that these define the intersection of an N-hypersphere (of radius $\sqrt{N}$) with a hyperplane, thus we naturally expect these points to be found on an (N-1)-hypersphere. Our purpose, however, is an operational one, namely a procedure for synthetically generating such points.

To achieve this, we will first proceed to a change of basis. We consider the orthonormal basis $\boldsymbol{U} = (\boldsymbol{U}_1, \ldots, \boldsymbol{U}_n)$ where

$$\boldsymbol{U}_1 \ = \ [\frac{1}{\sqrt{N}}, \ldots, \frac{1}{\sqrt{N}}]^T \tag{6.10}$$

and the rest of the basis is obtained by starting with $V_2 \ = \ [0, 1, \ldots, 0]^T, \ldots, V_N \ = \ [0, 0, \ldots, 1]^T$ and applying the standard Gramm-Schmidt procedure

$$U_i = \frac{W_i}{||W_i||}, W_i \ = \ V_i - \sum_{j=1}^{i-1} < V_i, U_j > U_j \tag{6.11}$$

Let $b_1, \ldots, b_n$ be $\boldsymbol{x}$'s coordinates in basis $\boldsymbol{U}$.

Because of equation 6.8 it holds that $< \boldsymbol{x}, \boldsymbol{U}_1 >= \frac{\sum_{i=1}^{N} x_i}{\sqrt{N}} = 0$. On the other hand, with $\boldsymbol{U}$ being an orthonormal basis, it also holds that $< \boldsymbol{x}, \boldsymbol{U}_1 >= b_1$. It follows that:

$$b_1 = 0 \tag{6.12}$$

Equation 6.9 in turn can be equivalently written as $< \boldsymbol{x}, \boldsymbol{x} >= N$, leading to:

$$\sum_{i=1}^{N} b_i^2 = N \tag{6.13}$$

Putting together equations 6.12 and 6.13 we end up with the (N-1) hypersphere equation expected, namely:

$$\sum_{i=2}^{N} b_i^2 = N \tag{6.14}$$

Generating the required $\boldsymbol{x}$ is then a two-fold process:

- First, we aim at producing uniform random points on the (N-1) hypersphere. To achieve this, we use the standard procedure for sphere point picking that consists in generating (N-1) Gaussian random variables $b_2^G, \ldots, b_N^G$ and producing (partial) $\boldsymbol{b}$ vectors as

$$[b_2, \ldots, b_N]^T = \frac{1}{\sqrt{\sum_{i=2}^{N} b_i^G}} [b_1^G, \ldots, b_N^G]^T \tag{6.15}$$

- Then, using the $\boldsymbol{U}$ basis coordinates of $\boldsymbol{x}$ obtained above and the $\boldsymbol{U}$ basis previously computed (equations 6.10 and 6.11), we recover the canonical coordinates of $\boldsymbol{x}$ as

$$x_i = \sum_{j=1}^{N} b_j * U_{ji} \tag{6.16}$$

**Generating normalized points constrained by distance**

We can further extend the above reasoning to obtain a refined primitive useful for workload generation, namely the ability of generating normalized points at a given distance $D$ from an existing *normalized* point $p = [p_1, \ldots, p_N]^T$. Indeed, this implies the required points $x$ further verifying the following equation:

$$\sum_{i=1}^{N} x_i^2 + \sum_{i=1}^{N} p_i^2 - 2 * \sum_{i=1}^{N} x_i * p_i = D^2 \tag{6.17}$$

where both $x$ and $p$ respect 6.8 and 6.9.

It follows that $x$ must further respect the following:

$$\sum_{i=1}^{N} x_i * p_i = N - D^2/2 \tag{6.18}$$

Note that the above is another hyperplane equation. Coupled with the restrictions on $x$ given by 6.8 and 6.9, it will thus unsurprisingly lead us to an $(N-2)$-hypersphere definition for the required $x$ points. Indeed, proceeding similarly as above, we will construct the orthonormal basis comprising:

$$\boldsymbol{U_1} = [\frac{1}{\sqrt{N}}, \ldots, \frac{1}{\sqrt{N}}]^T \tag{6.19}$$

and

$$\boldsymbol{U_2} = [\frac{p_1}{\sqrt{N}}, \ldots, \frac{p_N}{\sqrt{N}}]^T \tag{6.20}$$

Note that $< \boldsymbol{U_1}, \boldsymbol{U_2} >= 0$ and that both $U_1$ and $U_2$ are unit vectors, because of $p$ respecting 6.8 and 6.9. We generate the rest of the basis $\boldsymbol{U}$ as above, by employing the

Gramm-Schmidt procedure. As above, let $b_1, \ldots, b_N$ be $\boldsymbol{x}$'s coordinates in the basis $\boldsymbol{U}$. We then have:

$$b_1 =< \boldsymbol{x}, \boldsymbol{U}_1 >= 0 \tag{6.21}$$

because of 6.8, and also

$$b_2 =< \boldsymbol{x}, \boldsymbol{U}_2 >= \frac{N - D^2/2}{\sqrt{N}} \tag{6.22}$$

because of 6.18. Then, using 6.9, i.e. $< \boldsymbol{x}, \boldsymbol{x} >= 1$, we obtain the following equation for the remaining $b$ coordinates:

$$\sum_{i=3}^{N} b_i^2 = D^2 * (1 - \frac{D^2}{4 * N}) \tag{6.23}$$

Note that this is indeed the equation of an $(N-2)$-hypersphere as expected. To generate $\boldsymbol{x}$ points, we then sample $b_3, \ldots, b_N$ as described above and recover $\boldsymbol{x}$'s canonical coordinates accordingly.

### 6.5.3 Hardness assignment and number of points to add

Given the queries selected according to one of the procedures in the previous subsection and the input hardness values $[\alpha_1^\epsilon, ..., \alpha_n^\epsilon]$, we next proceed to (i) assigning queries to these hardness values and (ii) determining the number of points to be added so that the respective queries attain their matching hardness.

Hardness assignment is in part arbitrary, that is, any hardness value $\alpha_i^\epsilon$ can be matched with any query $q$ as long as the current $\epsilon$-hardness value (for the given $\epsilon$) of $q$ is lower or equal to $\alpha_i^\epsilon$. As we will detail in the next sections, additional constraints on the queries may be imposed (leading to a *ranking* function over queries), to improve the overall workload quality.

Once each hardness value has a corresponding query, we need to identify the number of points to add to the $\epsilon$-area of each query in order to achieve the target hardness. Let $x_i$ be the number of points to add for $\mathcal{N}^\epsilon(\boldsymbol{q}(\alpha_i^\epsilon))$ and $N_i = |\mathcal{N}^\epsilon(\boldsymbol{q}(\alpha_i^\epsilon))|$ is the current number of points in $q(\alpha_i^\epsilon)$'s $\epsilon$-area, we have the following linear system.

$$\alpha_1^\epsilon = \frac{N_1 + x_1}{N + \sum_{i=1}^{n} x_i},\ ...,\ \alpha_n^\epsilon = \frac{N_n + x_n}{N + \sum_{i=1}^{n} x_i} \tag{6.24}$$

Representing this linear system in matrix form, we have

$$(A - I)\boldsymbol{x} = \boldsymbol{b}, \tag{6.25}$$

where

$$A = \begin{pmatrix} \alpha_1^\epsilon & \alpha_1^\epsilon & ... & \alpha_1^\epsilon \\ \alpha_2^\epsilon & \alpha_2^\epsilon & ... & \alpha_2^\epsilon \\ ... & ... & ... & ... \\ \alpha_n^\epsilon & \alpha_n^\epsilon & ... & \alpha_n^\epsilon \end{pmatrix}$$

and

$$\boldsymbol{b} = [N_1 - \alpha_1^\epsilon N, ..., N_n - \alpha_n^\epsilon N]^T.$$

This linear system can be easily solved and it will tell us how many points to densify in the $\epsilon$-area for each selected query.

### 6.5.4 Densification Process

As we mentioned in Section 6.3, meaningful queries for effective index comparison should satisfy Equations (6.6) and (6.7). Given that for each summarization we can infer the $\epsilon$ areas that best characterize it using Equation (6.5), then we can assign an $\epsilon$ to each summarization. Given *any two summarizations* $S_1$ and $S_2$, a single query $q_1$ it should hold that:

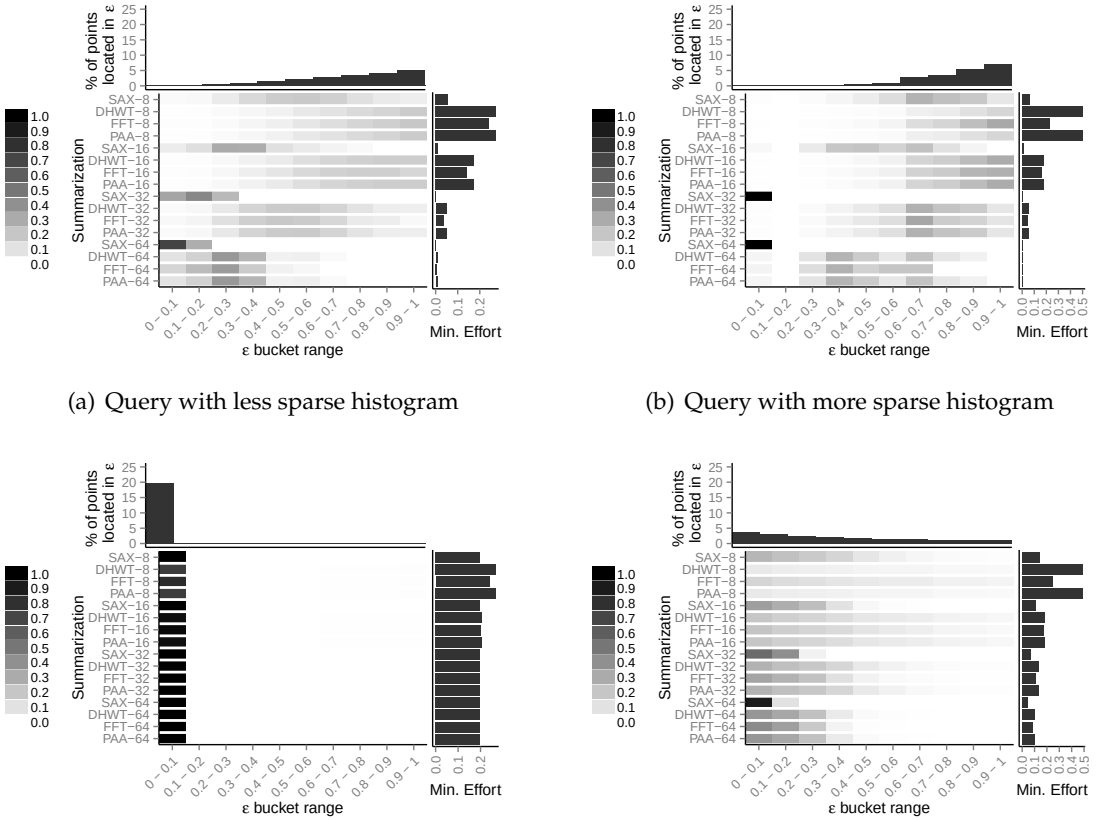
$$\frac{\mu^{S_1}(\boldsymbol{q}_1)}{\mu^{S_2}(\boldsymbol{q}_1)} = \frac{1 - TLB(S_1)}{1 - TLB(S_2)} = \frac{\alpha^{\epsilon_{S_1}}(\boldsymbol{q}_1)}{\alpha^{\epsilon_{S_2}}(\boldsymbol{q}_1)} \tag{6.26}$$

Since this should hold for any summarization (and as a result for every TLB and every $\epsilon$, it automatically makes both Equations (6.6) and (6.7) hold.

In this section we describe how to densify the $\epsilon$-areas for the selected queries for the above condition to hold. Our solution is named equi-densification. The key idea is that we add points in the right locations making sure that that our desired properties hold.

Additionally, in order to demonstrate why Equations (6.6) and (6.7) should hold we also consider two baseline candidate strategies for densification:

- RANDOM: randomly choosing points in $\mathcal{N}^\epsilon(\boldsymbol{q}(\alpha_i^\epsilon))$ and adding noise to create a new points;
- 1NN: adding noise to the query's nearest neighbor (ignoring all other points in its $\epsilon$-area).




(a) Query with less sparse histogram

(b) Query with more sparse histogram

(c) 1NN densified query

(d) Equi-densified query

Figure 6.6: Two randomly densified, one 1NN densified and one equi-densified queries on a 100,000 data series randomwalk dataset. Distribution of distances of all data series in the dataset on top, minimum effort for each summarization technique on the right. Heat maps represent the amount of points that are part of the effort located at the corresponding bucket of $\epsilon$.

To demonstrate the different densification strategies, we generate queries of hardness 0.2 ($\epsilon = 1.0$) for a dataset of 100,000 data series. This $\epsilon$ allows us to test less tight representations with *TLB*s as low as 0.5. To evaluate the effort for every query, we use four standard data series summarization techniques (SAX, FFT, DHWT, PAA) at various resolutions, ranging from 8 to 64 bytes per data series. The data series are of length 256, and for each summarization we measure the minimum effort required.

#### EQUIDENSIFICATION

As discussed in Section 6.3, we want to ensure that the hardness points are distributed as uniformly as possible within the $\epsilon$-area corresponding to each possible *ATLB* value. This ensures that we capture the subtle differences for various summarizations. To this end, we propose equi-densification that aims to distribute the extra points we need to add in such a way that buckets that are originally almost empty get a large number of points, and buckets that are almost full get a small number of points.

In order to achieve this, we bucketize *ATLB* values, and accordingly the $\epsilon$ values are bucketized (in a non-uniform way). For each *ATLB* bucket we want to make sure there is an equal amount of points we achieve this by placing points at the desired locations. This action is done by creating linear combinations of points located within and outside of each *ATLB* bucket. This ensures the diversity of the generated data series, allowing us to control the location of the data points in the $\epsilon$-area, and also ensures that the resulting data series after z-normalization will fall in the desired location with high probability. This algorithm produces the desired result but has a high complexity as we need to exhaustively test various factors for the linear combination ranging from 0 to 1.

A query produced with equi-densification is depicted in Figure 6.6(d). The histogram on the top shows that the first few buckets have more points, while the last few buckets have less. This happens because $\epsilon$ is inversely proportional to *ATLB*, and as a result $\epsilon$ bucket ranges are small for large *ATLB* values and large for small *ATLB* values. For example for *ATLB* values in $[0.5, 0.6]$ the corresponding $\epsilon$ values are in $[0.67, 1.0]$ and for *ATLB* values in $[0.6, 0.7]$ the corresponding $\epsilon$ values are in $[0.43, 0.67]$ As we can see in the heat map, the effort points are now evenly distributed in the $\epsilon$-areas. Note also that as the bounds of a summarization get worse, we need to increase the $\epsilon$ to include all points that contribute to the minimum effort.

Therefore, equi-densification achieves the desired result, accurately capturing the relative differences among different summarizations, and consequently leading to correct performance comparisons of indexes based on their $TLB$. We further validate this claim in the experimental evaluation.

#### Baseline methods

**RANDOM**. A naïve method to increase the hardness in an $\epsilon$-area is to choose random points from this area and add noise to them, thus producing the desired amount of extra points. A property of this method is that the original distribution of the points

will not change significantly. The problem with this method, however, is that for very good summarization methods (large *TLB* values), as we increase the number of points in the $\epsilon$-area, the minimum effort will not necessarily increase relatively to it. As a result, indexes with different *TLB* values might have the same effort to answer a query.

The result of a query generated with this method can be seen in Figure 6.6(a). The histogram at the top of the figure displays the distribution of the points in the densified $\epsilon$-area. As we can see the further away we get from the nearest neighbor, the more points we find at each area. The heat map in the center represents the locations of the points that contribute to the minimum effort, i.e., $L(\boldsymbol{x}, \boldsymbol{q}) \leq MINDIST(\boldsymbol{q})$. The color represents the portion of these points in the corresponding bucket of $\epsilon$. Finally, the vertical graph on the right side represents the minimum efforts of the different summarization methods.

As expected, the results show that crude summarizations (SAX-8, DHWT-8, FFT-8, PAA-8) that use less bytes for representing the data series have much larger minimum efforts. From the plot we could infer that a significant portion of points that contribute to minimum effort may not be included by this $\epsilon$-area. On the other hand, fine summarizations (SAX-64, DHWT-64, FFT-64, PAA-64) are well captured by this $\epsilon$. Actually, we only need $\epsilon = 0.6$ to capture all points contributing to the minimum efforts. With the histogram on the top, it is easy to see that the minimum effort is related to the distribution of points in the original space. For example, while the heat map for FFT-64, DHWT-64 and PAA-64 spans a larger range of $\epsilon$ values, their minimum effort is not much greater than that of SAX-64, which spans a much smaller range. This is because, as we can see in the histogram at the top, there is a very small amount of data within $\epsilon = 0.5$ and it does not increase too much as $\epsilon$ increases. This situation is more pronounced with another query example shown in Figure 6.6(b), where the distribution of points in the $\epsilon$-area is even more skewed.

**1NN**. Another naïve method for increasing hardness in the $\epsilon$-area is by just adding noise to the query's nearest neighbor itself. This will force all summarizations to make (almost) the same effort, as the area very close to the nearest neighbor is now very dense and all the rest of the $\epsilon$-area is very sparse. In this case, all efforts for all summarizations are almost identical. An example 1NN densified query is shown in Figure 6.6(c).

## 6.6 Experimental Evaluation

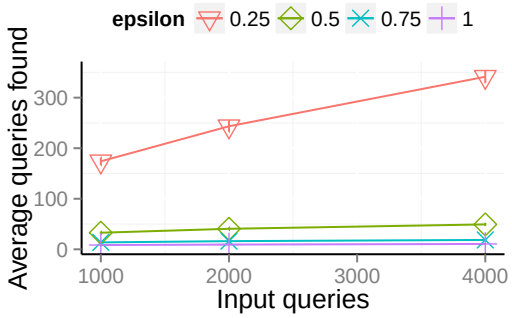
In this section, we provide an experimental evaluation of the proposed method. We generate query workloads on the three datasets in Section 6.4 using our method described in the previous section. All our datasets contain 100,000 data series with length 256. Given a set of desired hardness values, $\epsilon$, and the densification mode, our method produces a new dataset that is the original dataset with extra points, and a set of queries that forms the workload that matches the desired hardness values.

We performed three sets of experiments. The first investigates the amount of non-interfering queries we can find for each dataset. The second is intended to compare the three different densification methods with regards to the minimum effort of various common summarization techniques, i.e., PAA, FFT, DHWT and SAX. For each one of the summarizations, we used 8, 16, 32 and 64 bytes to represent each data series. In the third set of experiments, we used two real world indexes, *i*SAX 2.0 [31] and the R-Tree [68] using PAA as a summarization method. This last experiment aims to show the impact of our benchmark on these indexes compared to choosing random points from the dataset (queries are left outside of the indexed data). A comprehensive experimental comparison of various data series indexes is out of the scope of this study and is part of our future work.

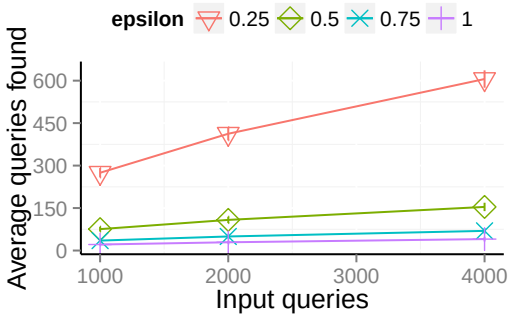
### 6.6.1 Non-interfering Queries

In this experiment, we used 100,000 data series from each dataset as the indexed data, and 100,000 data series as sample queries. We generated sets of 1,000 (100 sets), 2,000 (50 sets) and 4,000 (25 sets) queries, and run our non-interfering queries discovery algorithm on each one of them for $\epsilon \in \{0.25, 0.5, 1.0\}$. Our algorithm evaluates each set of queries against the corresponding dataset, and we report the average number of queries found per dataset, query set size and $\epsilon$, as well as the corresponding error bars.

The results are depicted in Figure 6.7 (error bars are very small). We observe that for RANDWALK and DNA, using a large $\epsilon$ only allows us to find an average of 7-10 non-interfering queries, and as $\epsilon$ decreases we can find up to 300-600 queries. For the EEG dataset, the data distribution allows us to find a much higher number of non-interfering queries, which is in the order of thousands. Note that since we are mainly interested in generating queries with high hardness values, we do not need too many queries. Furthermore, the constraint $\sum_{i=1}^{n} \alpha_i^{\epsilon} \leq 1$ restricts the number of hard queries that we could produce for one dataset. For example, we can generate 5




(a) RANDWALK dataset




(b) DNA dataset


(c) EEG dataset

Figure 6.7: Number of independent queries found for each dataset for various input sample query sizes.

queries of hardness 0.2, but only 2 queries of hardness 0.5 and 0.5, respectively. Since this number of queries is small for a comprehensive benchmark, the solution is to use multiple datasets with corresponding query workloads; even in the case of $\epsilon = 1.0$, we can run our algorithm 100 times to get 100 different output datasets with at least 3 different queries each, for a total of 300 queries.

### 6.6.2 Densification Mode

In this experiment, we generated 3 different queries with a hardness of 0.2 for each one of them ($\epsilon = 1.0$). For each query we used a different densification method. Our goal is to measure how well the different densification methods capture the relative summarization errors of different summarization techniques. We use $1 - TLB$ as the summarization error for each technique. This number intuitively captures how far

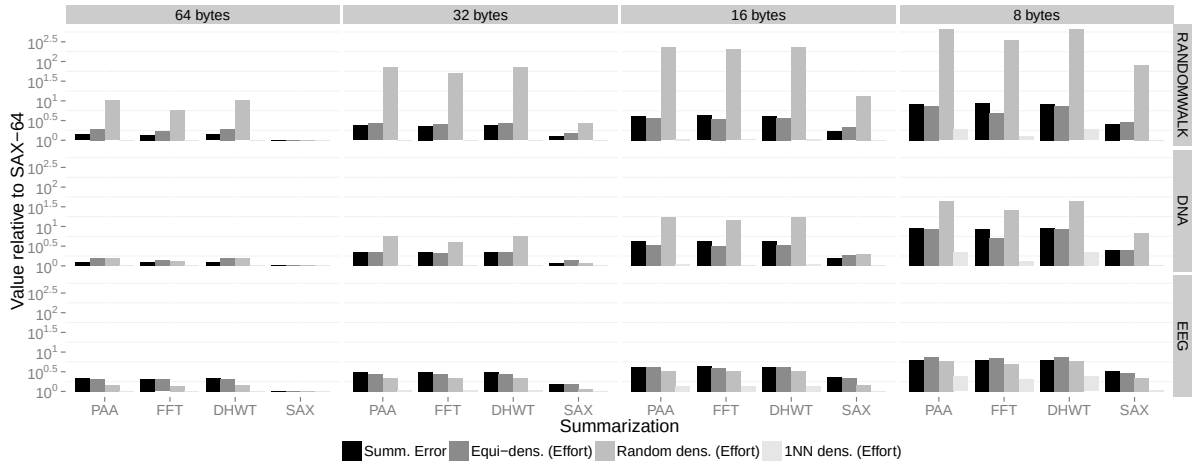


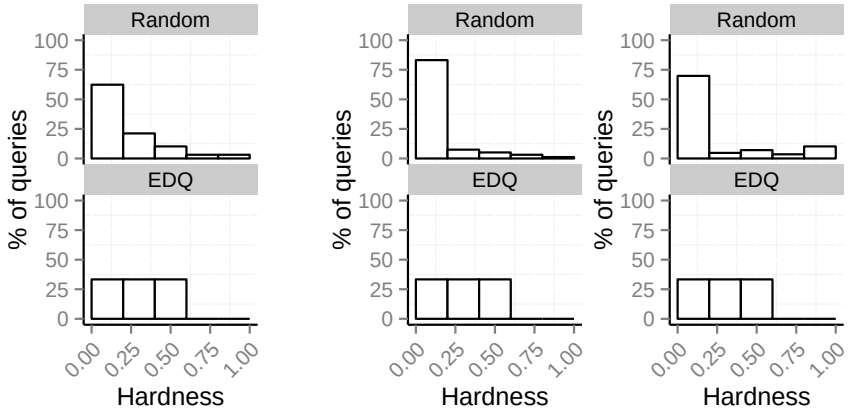

Figure 6.8: Minimum efforts for different summarization techniques at different resolutions (8-64 bytes) representing 256 point data series, compared to the summarization error (1-*TLB*). All the values have been normalized against SAX-64 which was the overall tightest summarization method.

the lower bound of a summarization is from the true distance. We report the relative summarization errors in the results (normalized by the smallest summarization error). In our experiments, the summarization with the smallest error was SAX (64 bytes). The *TLB*s for each summarization were computed by comparing the distances to the lower bounds for 100 random queries against all the other points of the dataset, for each of the datasets we generated.

Figure 6.8 shows the average relative summarization errors for each dataset (averaged over the 100 different benchmarks generated). The results show that 1NN densification results to almost equal effort for all summarizations, while random densification tends to over-penalize bad summarizations and favor good ones. Both situations are not desirable and cannot be useful. In contrast, equi-densification has an effort much more closely related to the summarization error across all datasets. As a result, equi-densification well captures the actual pruning power of each summarization and does not over-penalize or under-penalize any of the summarizations.

### 6.6.3 Case Study on Actual Indexes

In our last experiment, we generated 85 datasets with 3 equi-densified queries corresponding to each one of them, which we will refer to as EDQ, and 3 additional queries (per dataset) that were randomly selected from the input queries sample without any



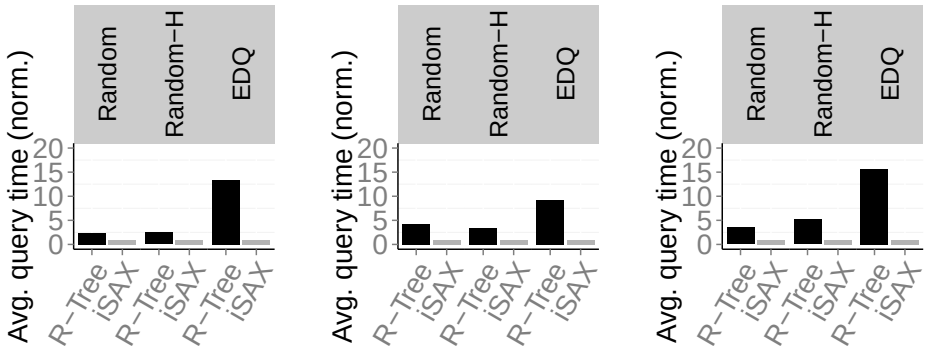

(a) RandWalk (b) DNA (c) EEG

Figure 6.9: Histogram of hardnesses for $\epsilon = 1.0$.

densification. The 3 EDQ queries have hardness values of 0.1, 0.3 and 0.5 ($\epsilon = 1.0$). We generated 510 queries in total, half of which were random and half equi-densified. Our goal for this study is to demonstrate the qualitative difference of using our query workload versus a workload of randomly generated queries.

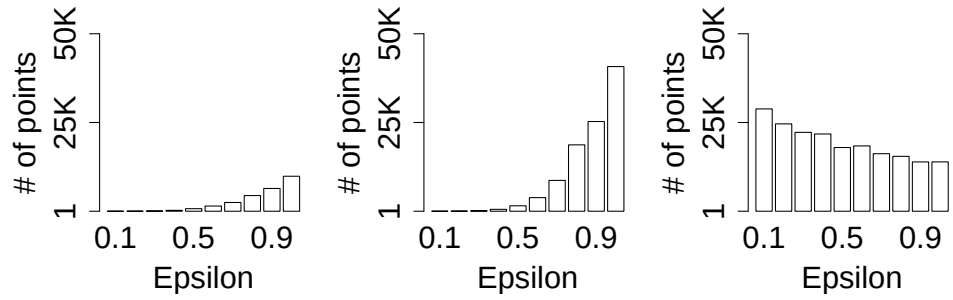
Figure 6.9 shows the histograms of the distribution of the hardnesses for the queries on each workload for every dataset for $\epsilon = 1.0$. Again, the random workloads are concentrated on easy queries with only a very small number of hard queries. On the contrary, the EDQ workload has been designed to produce queries of varying hardness values, and as a result their histograms contain equal number of queries in the 0.1, 0.3 and 0.5 bucket. This confirms that our method produces queries with desired properties.

In order to specifically evaluate the effect of hard queries, we further split the random workload into two sets, resulting in 3 different workloads: Random, where we use all the randomly selected queries, Random-H, where we only use queries with hardness larger than 0.5, and EDQ generated by our method. We indexed all three datasets with both *i*SAX [165] and R-Trees [68] with PAA [93], and measured the average query answering time per workload. Figure 6.10 illustrates the normalized query answering time. The results show that when the Random workload is used, queries are on average easy, and consequently, the two indexes seem to have similar performance. The same observation also holds when only the hard queries are selected using Random-H, indicating that simply selecting the hard queries of a randomly generated query workload cannot lead to a good query workload.




(a) RANDWALK (b) DNA (c) EEG

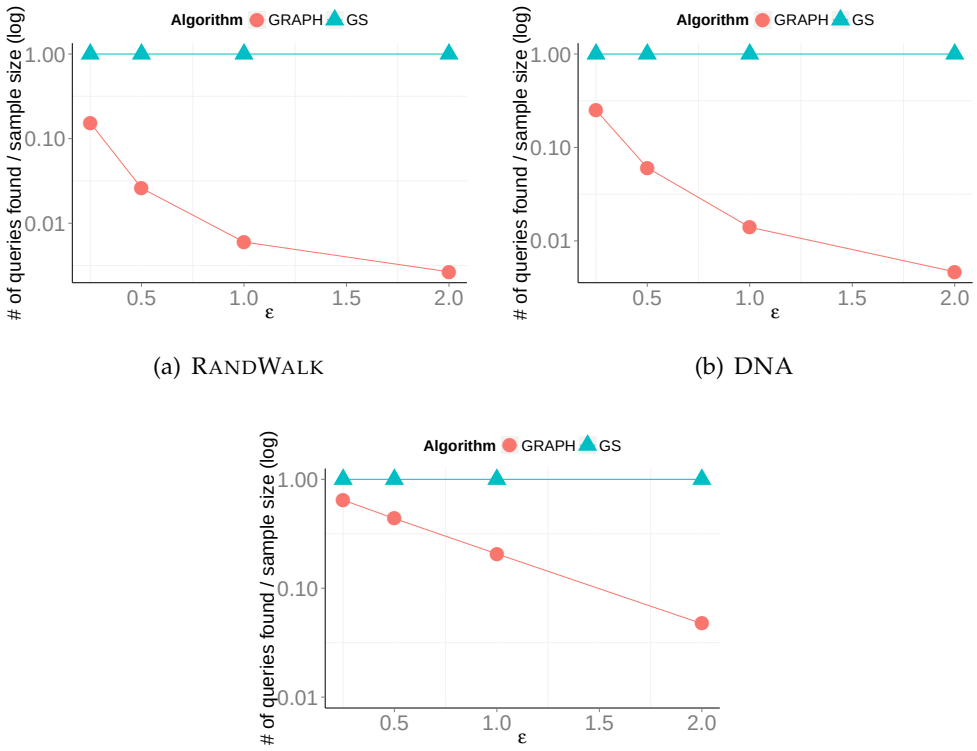
Figure 6.10: Average query answering time comparison between *i*SAX (256 characters, 16 segments) and R-Tree (PAA with 8 segments) normalized over *i*SAX.




(a) Random (b) Random-H (c) EDQ

Figure 6.11: Distribution of points in $\epsilon = 1.0$ area for 3 types of queries for RANDWALK.

The real difference comes when the workload becomes harder using the EDQ workload. In this case, the differences between the indexes become more prominent. The reason behind this can be intuitively seen in Figure 6.11, where we plot the distribution of the distances to the query's nearest neighbor in the $\epsilon$-area for the three different workloads. We can see that with random queries, (Random and Random-H), the vast majority of the points are located towards the large $\epsilon$ values. The difference between Random and Random-H is just on the number of points in each bucket. As we discussed earlier, such a distribution of points cannot capture the relative *TLB* of different indexes, as there are fewer points in small range to the true answer and many more points in larger range. On the other hand, the distribution of EDQ is very different from the others, which ensures there are roughly equal number of points for the corresponding *ATLB* bucket.



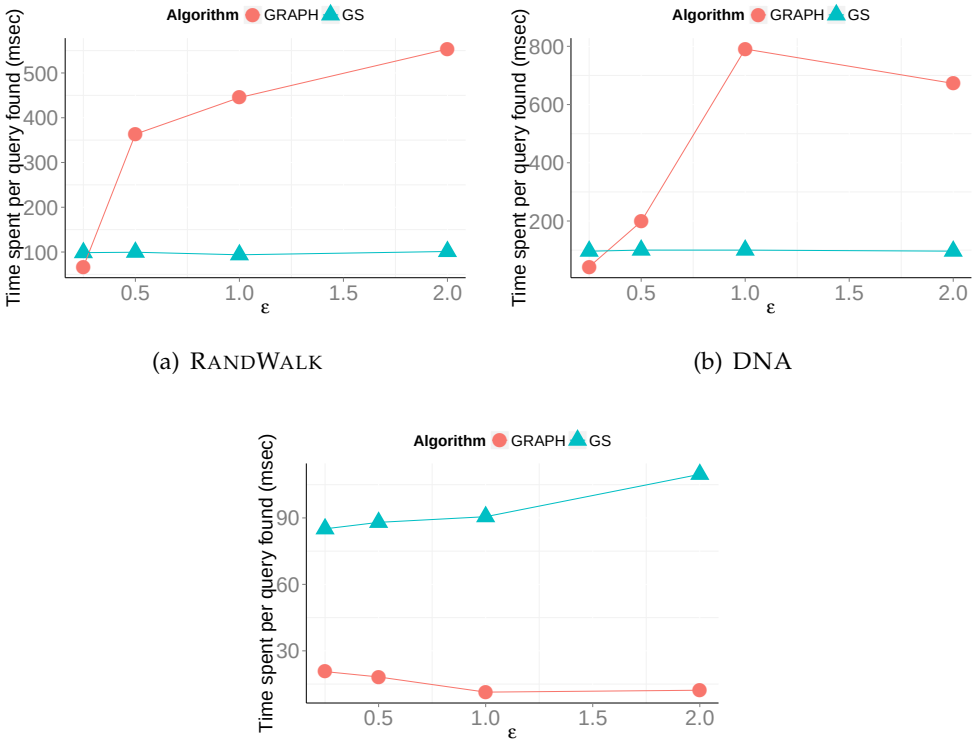

(a) RANDWALK (b) DNA

(c) EEG

Figure 6.12: Number of non intersecting queries found using both methods.

### 6.6.4 Improving the Amount of Queries Found using Synthetic Nearest Neighbors

As we have seen in Figure 6.7 there is only a limited amount of non-interfering queries that we can find. The amount of queries becomes even less as the $\epsilon$ increases, to the point that for most of the datasets we can only find a handful of non-intersecting queries, even when the sample size is large. In Section 6.5.2, we have presented a technique that allows us to generate new nearest neighbors for each one of the queries in our sample size. These nearest neighbors are placed in the appropriate distance, such that they create non-intersecting areas. Moreover, they are generated in such a way that they are already normalized. In Figure 6.12, we plot the ratio of non interfering queries found in a sample of 1500 queries and dataset sizes of 100,000 data series. We include the ratios for both the old graph based method (named as GRAPH in the plot), and the new Gram-Schmidt based method (named as GS in the plot). As we can see, using the old method, even in the case of very small $\epsilon$ (hardly useful in practice), we are able to use just above 10% of the queries from the sample for RANDWALK and DNA. For the case of larger $\epsilon$ values, we can use less than 1% of the sample size for




(a) RANDWALK (b) DNA

(c) EEG

Figure 6.13: Time spent per query found by each method.

these two datasets. Using the EEG dataset, nevertheless, we can use a larger number of queries but only for a very small $\epsilon$. As a result, for all the datasets, when an $\epsilon > 1.0$ is used, we can hardly ever use more than 10% of the queries. On the contrary, for the case of our Gram-Schmidt based method, we are able to select 100% of the sample size as queries, in all cases and all datasets.

### Time Complexity

In regards to time complexity, the overall time spent in order to solve the Gram-Schmidt equations is much larger than the time spent for providing an approximate solution to the graph problem. On the contrary, the amount of queries found using Gram-Schmidt is much larger. If we normalize the time spent over the number of queries found, it becomes clear that the Gram-Schmidt based method is much more efficient. This means the time spent per query found is much less than what is required by the graph based approach. This can be seen in Figure 6.13, where we plot the time per query found for both methods and different $\epsilon$ values. It is clear that in most cases, Gram-Schmidt

outperforms the graph based method. In the case of the EEG dataset, we are able to also generate many queries also using the graph based method, as a result the normalized time of the graph based method is less. However, both methods need less 100 milliseconds per query.

## 6.7 Summary

In this chapter, we focused on the problem of how to systematically characterize a data series query workload, and subsequently, how to generate queries with desired properties, which is a necessary step for studying the behavior and performance of data series indexes under different conditions. We demonstrated that previous approaches are not viable solutions as they are biased toward easy queries. We formally defined the key concept of query hardness and conduct an extensive study on hardness of a data series query. To solve this problem, we described a method for generating data series query workloads, which can be used for the evaluation of data series summarizations and indexes. Our experimental evaluation demonstrates the soundness and effectiveness of the proposed method. Finally, since our baseline method is able to generate only a limited amount of queries, we presented an additional method, which is able to computationally place artificial nearest neighbors. This allows us to generate an unlimited number of queries.

# Chapter 7

# Meta-data Enriched Data Series Exploration

In the previous chapters we concentrated on optimizing interactive similarity search for large data series collections. In this chapter we present a Business Intelligence scenario that allows analysts to perform principled monitoring of meta-data enriched data series. Specifically, we present a method that allows analysts to monitor business indicators in a goal-oriented way. This has been a motivating scenario for our work, as such kinds of real time business intelligence applications are in need of performing efficient and interactive similarity search [200]. Such operations aim at speeding up clustering algorithms, for identifying correlated market segments and indicators, as well as for identifying outliers that could represent business threats or opportunities.

One of the future directions that we plan to investigate is that of integrating our index structures in applications such as the one we present in this chapter. This is of high relevance to various companies that are deluged by vast amounts of meta-data enriched data series. For example, Wal-Mart generates 2.5 PetaBytes (PB) of transaction data each hour [192]. Since all these events are associated to specific timestamps, such data can be represented as time-series. Analysts are interested in monitoring these data, finding trends, clusters and outliers that are relevant to the organization's strategic objectives.

## 7.1 Monitoring and Diagnosing Indicators for Business Analytics

Modeling the business strategy has been shown to be useful both for understanding a business [15] as well as planning and guiding the activities within an enterprise [87].

Most enterprises represent the business strategy in a textual fashion, captured in the business plan. Then, once the business plan has been established, the business intelligence system of the organization helps to monitor business performance by means of Key Performance Indicators (KPIs) [128]. For example, an organization may have the goal of "Increasing revenue" by achieving "Increasing market share". These goals could be monitored by the KPIs, "Revenue" and "Market share".

Traditionally, organizations are interested in monitoring these KPIs in order to identify unwelcome or unexpected situations, either positive or negative, that affect their goals. Currently, this kind of analysis is done by measuring how close or how far the values of KPIs are from their targets [128], defined in terms of an acceptable range. It is often the case, however, that these KPIs do not reflect the anomalies within the subareas that are being monitored by the KPI, since they represent high levels of aggregation. We refer to these sub-areas as *sub-markets*. For example, sales may have decreased dramatically in Trento, but this may not be noticeable by looking at national aggregates for Italy. Moreover when KPIs do deviate significantly from their target values it is mostly the task of an analyst to seek the reasons behind these exceptional events. Finally, such KPI deviations are only monitored in isolation, non-systematically, rather than within the context of the strategic model. As a result, the impact analysis for a deviation is limited.

Although current dashboards and scorecards allow users to analyze the data in detail, performing the monitoring and analysis processes manually can be a daunting task, since (i) the underlying data warehouse commonly contains multiple dimensions [96], thus making analysis a time-consuming process, (ii) identifying a significant event is challenging, due to the knowledge required to interpret the data for each specific part of the market, and (iii) explaining the results in the context of a strategic model is even more difficult, since it presupposes understanding how the results may affect every relevant goal and indicator.

In order to tackle these problems, we propose a semi-automated method for generating a set of monitoring and diagnostic queries from a strategic point of view, in order to (i) identify unexpected and/or unwelcome situations in the context of a strategic model, (ii) explain why these situations occur, and (iii) identify how they may affect other strategic elements. Furthermore, we show how all the steps in our approach can be applied to a real case by means of an illustrative running example based on the Europe 2020 framework, to be described in the following section.

Specifically, in our approach, we monitor groups of related indicators for identifying if certain goals are going to be achieved on time. Moreover, we are able to detect

outliers within such groups that could point us to anomalies that should be either corrected or confirmed, i.e., "Why Sales have dropped in Barcelona but not in Madrid?". Additionally, we are able to explain such outliers by focusing on specific areas of the data warehouse that are responsible for these anomalies. Finally, we consider that our approach could be applied to work with big data analysis techniques, thus allowing the results to be evaluated from an strategic perspective and helping users to understand what these results mean for the business.

The remainder of this section is structured as follows. Section 7.2 presents the background work, including an illustrative example. Section 7.3 describes an overview of the steps involved in our approach applied to our case study. Finally, Section 7.4 discusses the results and summarizes our work.

## 7.2 Background & Problem Formulation

It is generally accepted that Strategic Management leads to better results among businesses. This is concerned with the continuous evaluation and control of a business and the environment within which it operates; it assesses internal and external factors that can influence organizational goals, and makes changes to goals and/or strategies to ensure success. Strategic Management has been practiced since the '50s, thanks to seminal contributions by Alfred Chandler, Peter Drucker and others who emphasized the importance of well-defined objectives and strategies that together determine and guide organizational activities [119]. Specific analysis techniques have been developed to support such processes, including the strengths-weaknesses-opportunities-threats (SWOT) analysis technique widely used in practice [100].

SWOT analysis focuses on internal strengths and weaknesses, as well as external opportunities and threats that may facilitate/hinder the fulfillment of organizational objectives. Once such SWOT situations are identified, they need to be continuously monitored in real-time to assess the degree to which they occur and keep track of their evolution over time (monitoring). In addition, operational data need to be continuously scanned in search of emerging SWOT situations or unexpected data (outliers). Being able to identify such threats and opportunities requires systems able to process data as they evolve, as well as, algorithms able to discover trends and deviations hidden among multiple layers of aggregated information.

The most common asset used for monitoring goals by enterprises until now has been the Balanced Scorecard [87]. A scorecard integrates several high level KPIs in order to provide an overall view of the performance of the business. However, a score-

card only provides an aggregated view of the data and does not consider the relationships between indicators. In an attempt to provide deeper analysis, businesses also include several Dashboards [57] that provide more detail about each indicator. Despite this effort, the potential number of different sub-market aggregations make it next to impossible to ensure that no anomaly goes unnoticed.

In terms of business modeling, Strategy maps [86] provide an overall view of the strategy of the enterprise. However, they are not (i) completely formal and (ii) they do not provide an integrated view of the status of the business. Similarly, in other areas, such as Software Engineering, proposals as [167] have been defined in order to specify monitoring conditions over requirements. Yet, in most cases the monitoring task is still left to the user. In [28] the authors propose the Goal-Question-Metric approach in order to monitor software development. However, when the approach is applied to business environments, the result obtained is a set of KPIs for monitoring the business performance that suffer from the drawbacks presented in the introduction. Finally, In [97] the authors propose the Willow architecture for system survivability. It aims at making systems avoid, eliminate and tolerate faults. It is able to monitor fault sequences, their inter-dependencies, as well as fault hierarchies. Each fault sequence is modeled by a finite state machine (FSM) which is triggered by system events. When certain FSMs reach a fault state, action is taken. In [110] the authors propose a way to visualize the business strategy and the indicators associated with it in an integrated way. Furthermore, the authors allow the user to define its own KPIs and keep track of the dimensions related to the calculus of each indicator. However, this approach does not cover any monitoring aspects and this task is left to be covered by the users. Yet, although such systems work well for system survivability, their purpose is to identify events and perform actions accordingly, rather than monitor trends towards the fulfillment of a set of goals as well as their statuses.

In order to effectively monitor their goals, organizations are required to precisely define the Strategic Objectives they are trying to fulfill as well as the factors that affect them. As a result, data analysis within them should also be done in this direction, as analysts need to see the data that are related to their objectives and goals [14]. This means that objectives that organizations are trying to fulfill form also their analysis goals and targets. Towards this direction, there has been work on capturing these goals in a modeling language such as the Business Intelligence Model (BIM) [85]. This language can be paralleled to the Entity-Relationship Model, but with concepts such as 'goal', 'situation', 'indicator', 'process' etc. These kind of elements and their relations can be formally captured using BIM. Such models are used for manually monitoring

the degree to which certain situations hold and how they affect the goals of the organization. These degrees are captured by Indicators that are calculated on top of data warehouses. Moreover, thresholds are provided to mark whether a goal is fulfilled, partially fulfilled, denied, etc. [14]. Previous work, from which BIM draws inspiration, has been also done in Goal-Oriented Requirements Engineering [52, 174], influence diagrams [70, 147] and the Business Motivation Model [66]. The BIM model includes the following concepts.

- A **Goal** represents an objective of the business (e.g. Increase Market Share). Decompositions of goals can also be defined with AND/OR relationships.
- A **Situation** is designed to represent the results of the SWOT (Strengths, Weaknesses, Opportunities and Threats) analysis commonly performed in strategic management. They are internal or external situations that affect the fulfillment of the goals
- An **Influence** from a situation to a goal, is a logical influence that positively or negatively affects the goal's fulfillment depending on the degree to which a situation occurs.
- An **Indicator** is a measure that can be attached to a Situation or a Goal in order to measure the degree to which they occur. They model what is traditionally done through Key Performance Indicators (KPI) in Management.

Such formal specifications of the Strategic Objectives, can also form the baseline for the analysis objectives for the data warehouses of large organizations. Using such formalizations, one is able to create systems that assist the users perform goal-oriented analysis of the multi-dimensionally organized data series in the data warehouse.

Additionally a set of related works are the ones related to Sentinels [114] and the OODA concept [113]. The OODA concept describes the loop of Observation (are the data normal?), Orientation (what is wrong?), Decision (user analysis) and Action (course of action) on top of a set of KPIs. Sentinels are causal relationships between KPIs mined in the data warehouse, which are used to trigger early warnings, thus helping the analyst perform OODA cycles efficiently. Such sentinels could be mined and integrated in a Strategic Model as situations that can affect goals.

#### Modeling the Business Strategy: An Illustrative Example

In this subsection we describe the basic elements within the business strategy in our approach, by means of an illustrative example based on the Europe 2020 framework.

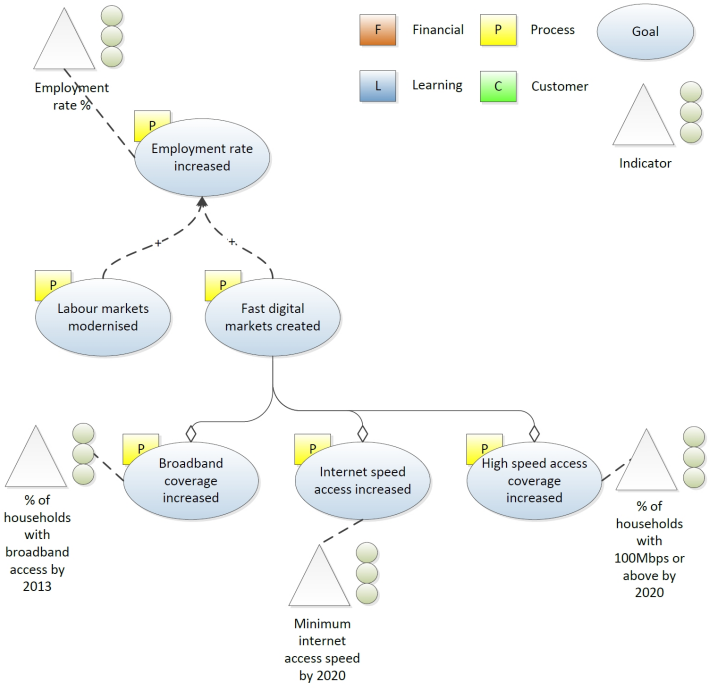



Figure 7.1: Excerpt of the European 2020 strategic model

The Europe 2020 framework aims to specify a set of strategic goals to be met by the European Union (EU) by 2020[1]. Some of these goals have clear target values and indicators established, allowing the EU to monitor their performance and be aware of deviations from the initial plan. Additionally, as the EU is integrated by several countries, each one of them with its own characteristics such as population, industry, etc., each country has its own particular objectives. As a result, under-performers may be compensated by over-performers, resulting in a high level indicator that correctly shows the UE is meeting its goal, without reflecting outliers that represent a potential threat. Furthermore, much like in traditional business plans, the descriptions provided in the framework only highlight a handful of relationships between goals, with no claims of completeness or consistency. We can see an example of these goals, indicators and the relationships between them in the Europe 2020 employment axis represented in Figure 7.1.

In order to model the business strategy, we make use of a simplified version of the Business Intelligence Model (BIM) [74]. According to this metamodel, the elements involved in the Europe 2020 framework are as follows:

[1]For more information see: ec.europa.eu/europe2020/index_en.htm

First we have **Goals**, which capture the objectives of the organization to be achieved. There are three kinds of goals: Strategic (long-term), Operational (medium-term) and Tactic (short-term). Within the Europe 2020 framework we have Strategic Goals that define the axis of the strategy, such as "Employment rate increased", "Expenditure on R&D increased", or improve "Environmental care". As some of these goals can not be monitored directly, they are refined into additional strategic goals. For example, "Environmental Care" is refined into diminishing "Green house gas emissions", increase the "Share of renewable energy in gross final consumption", and diminish "Primary energy consumption". In order to achieve these high-level goals, a set of Operational Goals, that influence the strategic goals, are defined. Examples of operational goals are "Fast digital market created" or "Individual skills developed". Finally, since Europe 2020 is a long-term plan, it includes no Tactic goals.

Second, we have a set of **Indicators** that monitor the performance of Europe 2020 goals, and alert about deviations in the targeted values. Some of the indicators included in the Europe 2020 strategy are "Employment rate %", "Early leaver % between 18 and 24", or "Index of Greenhouse gas emissions". Each of thse indicatorsr presents a target value (value to be achieved), a threshold (margin between good and bad performance), a current value and a worst value. According to these values we can analyze how much we are deviating from our targets. In addition to these attributes, in this work we extend BIM with two additional attributes. The first one is the "time to target" often employed in scorecards and required in order to perform time series analysis. This attribute describes how much time is left to achieve the expected target value. The second attribute is the refresh rate of the indicator, which is required for monitoring purposes, and describes how often the indicator should be monitored.

Third, we have **Situations**, which describe external or internal influences that may affect the business strategy and its goals, positively or negatively. An example situation would be "Economic Crisis". However, since the Europe 2020 framework does not explicitly mention this situation (it is managed by the EU on its own), it has been omitted from the model.

Finally, we add the concept of **Strategic query** over the model. A strategic query formalizes a goal, allowing to check if it is achieved or not. A simple query for "Employment rate increased" goal would be $EmploymentRate >= 75\%$. However, these queries can be made more complex, including subtargets $\forall C_i \in Countries, EmploymentRate_C >= Target_C$ and trends $EmploymentRate_{2020} >= 75\%$, or involving multiple goals [74].

By modeling these elements, and by associating KPIs to business goals, we obtain a clear view of the strategy that our business is following, as well its current status.

### 7.2.1 Monitoring a Business Strategy

Monitoring a business strategy by relying only on KPI information (current vs target values, time left) can conceal lurking problems, especially when KPIs are highly aggregated. Therefore, it is necessary to issue queries that monitor the evolution of both KPIs and sub-areas within these KPIs periodically.

For example, consider again the case of Europe 2020. The aggregated indicator for Employment Rate currently shows a deviation of less than 7% compared to its target value. However, is this deviation distributed equally among each country? It may be known that in some countries unemployment rates are increasing, leading to increasing deviations from their targets. However, the community as a whole may not be aware of these deviations until they become problems that threaten the global target.

In order to adequately monitor these situations, we need to pose strategic queries. For example, in the case of Europe 2020 a manager could pose queries described in natural language such as "Is it expected that we meet our Employment Rate goal?" This query would derivate into several other queries like "Are there other countries displaying a similar pattern to those that are struggling?" "What countries are close to their own sub-targets?" The information gathered from these queries helps to monitor the status of the strategic goals and identify potential problems arising, even if the aggregated indicator is not accurately reflecting them. However, it is often the case that anomalies are latent and thus, the number of monitoring queries that can be posed increases exponentially, thus complicating the monitoring task. Therefore, in order to restrict the search space it is necessary to determine two aspects:

1. **Dimensionality**. What are the relevant dimensions that we are interested in analyzing in detail? In Europe 2020, we are interested in finding anomalies within Country and Time dimensions, whereas if we were analyzing the evolution of sales we might be interested in Customer Segments. These relevant dimensions are not necessarily those restricting the calculation of the indicator to be monitored.

2. **History**. What are the relevant periods of time for the queries of interest? In the case of Europe 2020, we may be interested in analyzing only the period after the start of the economic crisis, or we may actually want to consider the data before the beginning of the crisis.

Finally, once these aspects have been determined, we need to transform strategic

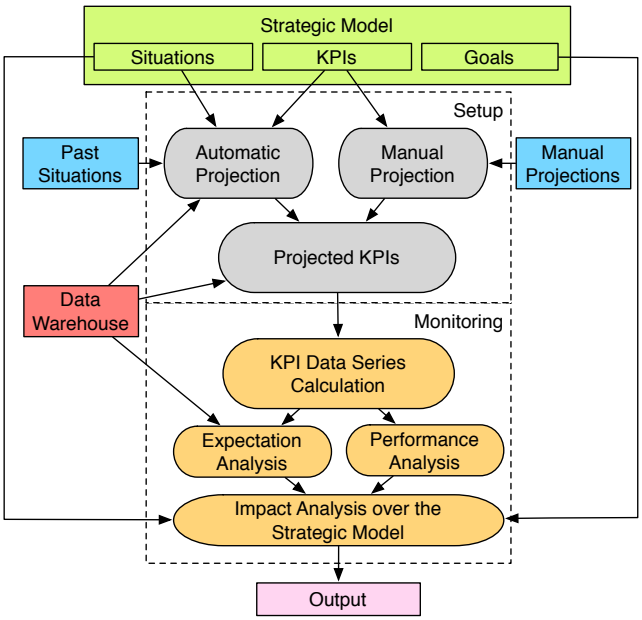



Figure 7.2: Overall view of the monitoring process

queries into one or more data warehouse queries that monitor the strategy and identify potential outliers and anomalies that can be highlighted for the analyst.

## 7.3 Proposed Approach

In this section we describe our proposal for monitoring the business strategy and detecting anomalies. An overall view of our process can be seen in Figure 7.2. First, we gather input from the user. This input is composed by the set of goals, situations, and KPIs to be monitored. Then, for each KPI, we gather the relevant dimensions from the data warehouse that should be considered (Dimensionality) and the relevant period of time (History). As a data warehouse may contain several dimensions, performing a search on every possible combination can be very costly, thus the initial knowledge from the user can speed up the analysis process. After gathering the input from the user, we proceed to the setup step. The setup step calculates the necessary data for the analysis. Specifically, we identify the target values for each sub-market to be considered. Afterwards, we proceed to the monitoring step. In the monitoring step, a set of algorithms analyze the data and identify the existence of deviations and outliers, whether in the aggregated values or in any of the sub-markets.

Our process requires us to be able to decompose business strategies into components, and specialize them to lower level sub-markets. All of the above in an automated manner that will assist analysts to monitor their strategic goals by pointing out:

- The performance for each goal based on the targets for their KPI values. Either at the highest level of aggregation, or at different sub-markets. *For example we might need to know that a specific goal has failed for Western Europe, even if it has succeeded for Europe in overall*.
- The KPIs that demonstrate unexpected behavior with regards to previously correlated markets, or their parent market segments. *For example we might need to know if Italy is following a different trend than the rest of Europe, or if Italy is following a different trend than Greece, even though they were correlated in the past*.

Based on the above requirements, we define two different kinds of diagnostics:

- **Performance diagnostics:** How are we doing with regards to a goal, based on the KPI value and our targets?
- **Expectation diagnostics:** Is the current value/trend expected based on the data in other parts of our data warehouse?

After providing an overall view of the process, in the following subsections we describe our process in detail using the Europe 2020 framework, previously introduced in Section 7.2.

### 7.3.1 User preferences

The first step in our process is gathering user preferences. Recently, employment rates in certain countries as well as the importance of education have been a hot topic. The current Employment and Education aggregated indicators do not show extreme deviations, and we have no knowledge about the potential influences among them. Therefore, as users, we will choose to focus on the Employment and Education axis from the Europe 2020 framework. The description of these axis is as follows:

- **Employment** axis has the target of achieving a 75% aggregated Employment Rate in the whole EU. Furthermore, each country has assigned its own sub-targets, such as 74% for Spain, or 67% for Italy. This axis is modeled in Figure 7.1.

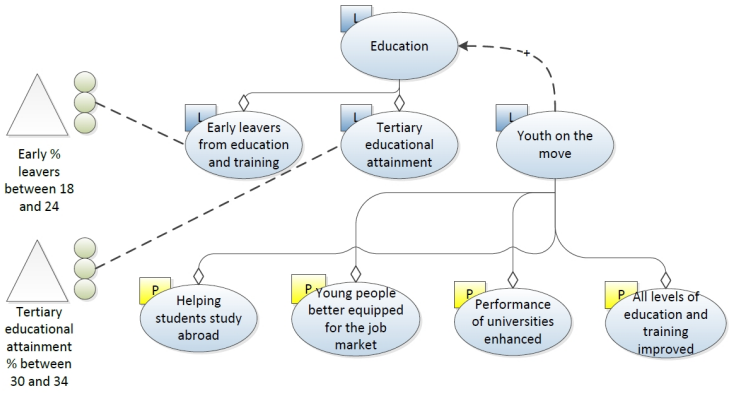



Figure 7.3: Education axis within the Europe 2020 strategy

- **Education** axis is subdivided into two main indicators. The first, measures the rate of **Early leavers from education and training**, and its target value is set to 10% or less of abandonment rate for people between 18 and 24 years. In addition to having different sub-targets for each country, the comparability of this indicator is restricted over countries and time, due to different implementations in the way of measuring its value. Second, the EU aims to achieve a **Tertiary educational attainment** rate of 40% or more for people between 30 and 34 years, with each country having its own sub-target. This axis is modeled in Figure 7.3.

Furthermore, each of these axis is supported by one or more initiatives planned by the EU commission, and grouped into pillars. Each initiative represents a course of action to be followed in order to achieve the strategic and operational goals, and may include milestones and sub-indicators to measure their progress. The pillars that support Employment and/or Education are:

1. **Smart Growth** pillar includes the initiatives "Creating a single digital market based on fast and interoperable applications", focused on increasing internet speeds throughout Europe, and "Youth on the move", focused on improving individual skills and foster student mobility.

2. **Inclusive Growth** includes the initiatives "Creating an agenda for new skills and jobs", focusing on helping people acquire new skills and modernizing labor markets to raise employment levels.

In order to support the analysis of this strategic model we require a data warehouse. A data warehouse stores information in terms of facts [96], and is structured according

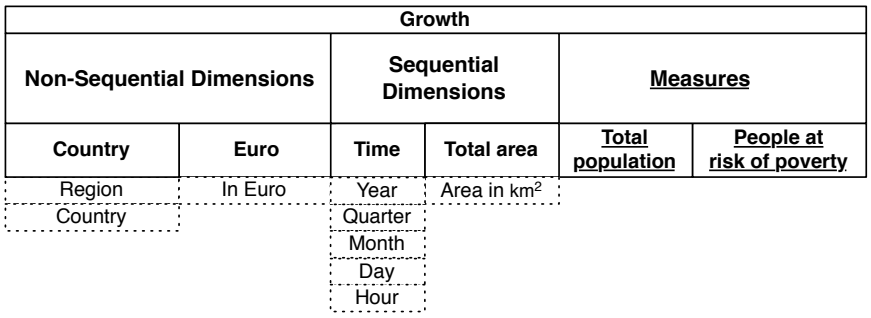



Figure 7.4: Europe 2020 Framework data warehouse

to a Dimension Schema $\mathcal{D}$, a Measure Schema, and a Fact table. The Dimension Schema defines a set of dimensions that provide context information. The Measure Schema contains all the measures available in the data warehouse. These are real numbers on which we can apply functions and aggregate them over the different dimensions defined in the Dimensions Schema. Finally, the Fact table stores the fact data. On top of a data warehouse, analysts can define KPIs, by combining aggregations (KPI terms) into complex formulas and by assigning target values. Further on, these KPIs can be restricted to various sub-dimensions e.g. specific countries, via KPI-Restriction operations that drill down in the data warehouse, for all the terms of a KPI.

An example Dimensions Schema is that of Figure 7.4, where the non-sequential dimensions are $\mathcal{D}_{NS} = \{Country,\ Euro\}$, the sequential[2] are $\mathcal{D}_S = \{Time,\ Total\ area\}$, and we have the following hierarchies $I = (i_{Country} = (Region, Country),\ i_{Euro} = (In\ Euro),\ i_{Time} = (Year, Quarter, Month, Day, Hour),\ i_{Total\ area} = (Area\ in\ km^2))$.

Note that we make a distinction between sequential and non-sequential dimensions, such that we are able to monitor trends of measures over the values of sequential dimensions, and compare these trends among different parts of the data warehouse. The most intuitive sequential dimension is that of Time, where analysts need to compare how the values of various indicators fluctuate over the course of time. Other examples can be relatively stable dimensions that can not be considered measures, such as the area of a country, where analysts are interested on monitoring the trends by using the area of each country in the horizontal axis.

[2] A sequential dimension contains instances that have an order established

With these considerations, we gathered the data for our case study from the official source for Europe 2020 data, the Eurostat[3]. Eurostat provides data in the form of tables for each indicator both at aggregated EU level, as well as for each country. A subset of the data warehouse schema can be seen in Figure 7.4.

According to this schema, we choose to use both Time and Country dimensions for the analysis of Employment and Education indicators, and we include all the data available since year 2000 into the analysis.

### 7.3.2 Setup Step

After the user preferences step, we proceed to the setup step. During the setup step, if the user has not selected specific KPIs, we identify all the KPIs for each goal by scanning the strategic model. Then, for each KPI, we need to identify all the KPI target values for each of its sub-markets, in order to support **Performance Diagnosis**. This can be done by either scanning a Knowledge Base where these projections are predefined, or in case that it does not exist, use the data warehouse for adjusting these values for each specific sub-market. This is done based on the relative size of this sub-market with regards to the higher level aggregation. In our scenario, we want to analyze Employment Rate, where we can adjust the target value for specific countries by finding the ratio of the historic average employment for a specific country, over the European historic average. An issue that arises here is that there might be more than one parent markets when we have sub-markets that are restricted by more than one dimension or that have multiple hierarchies of aggregation. So we need to decide, for example, if it makes sense to compare Italy to the countries that are in Eurozone (Eurozone classification), to the countries that are in South Europe (Region classification), or to all the countries. One could consider all of them and average the ratios, or pick the one that makes more sense. In the following subsections we analyze these aspects in more detail.

#### KPI Value Segmentations for Performance Diagnostics.

Being able to produce sequences of observations for each KPI, and specialize them to various sub-dimensions or hierarchy levels, will give us the raw data. But we still need to address two important points in order to support Performance Diagnosis.

1. The first problem is that of missing values. There are cases where we have data

[3]http://epp.eurostat.ec.europa.eu/portal/page/portal/eurostat/home/

for all the years except one, or cases where we are interested on extrapolating our data for predicting future values based on the past trends. In such a scenario, we need to be able to either interpolate the data, in order to produce an approximation of a missing value, or if we are interested in forecasting future values, use a time-series forecasting algorithm to do so. This will help us in two directions:

- First, we will be able to create complete data series even if some of the intermediate values are missing.
- Second, we will be able to produce ahead of time (if the sequential dimension is time) performance analytics, and thus identify if a goal is going to fail or succeed in the near future.

2. The second problem is that of measuring the degree of performance success for different KPI-Restrictions. For example, given that a value $> 1\%$ of people in risk of poverty is bad, we would like to find out what this number is for other dimension restrictions e.g. per country, in/out of Eurozone, or even different years. This will help us automatically identify our target values for sub-markets as well as sub-periods for a specific high-level KPI, that is defined in a strategic model.

**Missing and Future Values.**

Initially, given a set of observations seen so far for a KPI, we are interested in forecasting its future values, such that we provide in-time insights. The values that compose each KPI can be seen as a training set, based on which a prediction model $p$ can be trained. This prediction model can then be used to forecast the observations that have not been recorded yet. For example, in Figure 7.5, we can see Employment Rate values projected into the missing years, in order to analyze if we will meet our goal.

**Definition 13 (KPI Data-Series Value Forecasting Query)** *Formally, given a KPI Data-Series* $ds = DS_{k_i}^{d_j}(i_s, i_e)$ *for a period* $i_s$ *to* $i_e$*, on a sequential dimension* $d_j$*, we are interested in training a prediction model* $p$ *in order to predict values of* $ds$ *for time points* $i_f > i_e$*.*

$$forecast(DS_{k_i}^{d_j}(i_s, i_e), p, i_f) \text{ returns } x \in \mathbb{R}$$

In the same spirit, we can define an operation that interpolates missing values within a certain range. In both forecasting and interpolation, queries can be answered

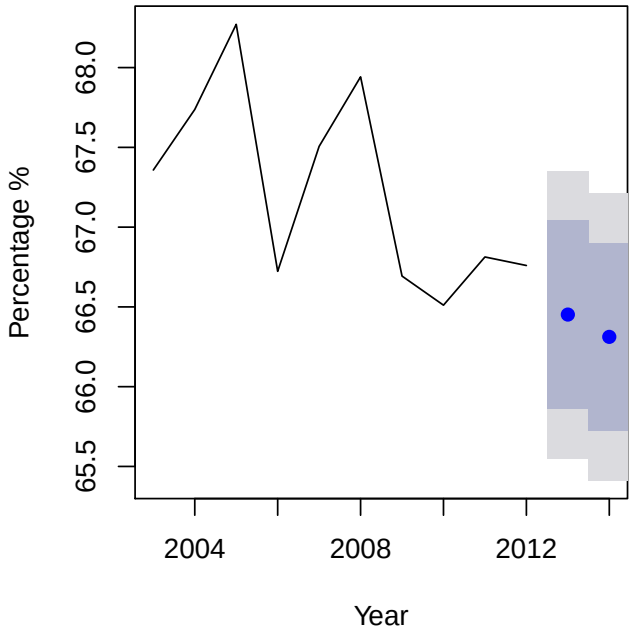



Figure 7.5: Total employment forecasting for EU

using well known statistics algorithms. Some common forecasting methods are those of moving averages, ARIMA models and the Box-Jenkins methodology [23].

**Projecting KPI Value Segmentations.**

As we stated earlier, being able to project target KPI Value Segmentations to KPI-Restrictions (sub-markets) is crucial for monitoring sub-markets. This is because we need to be able to monitor our expectations on a higher-level market aggregation (e.g. unemployment all over Europe) as well as for different sub-markets (e.g. unemployment in Italy, France, etc.) and not only in overall.

In order to do so we need to be able to extract the value segmentations for a KPI-Restriction in any given dimension-instance pair, either sequential or not. So for example, given that an Employment Rate of 75% is "good" for EU in overall, it is not trivial to project what is good to different sub-markets such as different countries, countries within Eurozone and more. Given this, we define a KPI Value Segmentation Projection Query that should be able to mine a database for identifying such KPI-Segmentation breakdowns.

This is a process that can be done either in an automatic, or in a manual way. This means that either the system must be able to project the value segmentation [105] to

lower level markets, for example by comparing market shares of previous years; or that the analyst has to manually specify a projection function that maps segmentations to the lower-level.

The most intuitive way of breaking down KPI segmentations is that of allowing the analyst to **manually define** them. These are two examples that demonstrate this case.

**Example 3 (Temporary Reforms: Manual Projection on the Time dimension)** *After applying some temporary economical reforms, we could specify that we expect a slight decrease in unemployment within a certain period. As a result, we would need to specify a 20% unemployment target for the first year, while for the second and third year we should have a 16% and 10% unemployment rate, even if our final target was 10%.* If we had not projected our expectations for all the years, we would have been failing during each year until we reached our target, even though this should not be the case.

**Example 4 (Development Level: Manual Projection on the Space dimension)** *In the same sense, one would expect that unemployment should be lower in more industrialized and developed countries than in developing ones. As a result we might have expected that the unemployment in a certain country should be lower than others over the years. This is the case for Europe 2020, where the Employment Rate target for each country is set individually.*

The second way for projecting segmentations is that of **automatically mining** the data warehouse. Given a database with values from previous years and a KPI. We can calculate, for every KPI-Restriction, the value of this KPI. Moreover, for each term of the KPI we can identify what is the market share of this term when compared to the "larger" parent market. This is an essential operation on data warehouses, where we calculate what is the contribution of a base-cell on a higher level aggregation. Since KPIs are complex functions, their values can not always additively calculate the higher level KPI. However, we are able to identify the contribution of each simple aggregation term of a KPI-Restriction to the general KPI. By feeding these ratios in the KPI calculation formula, we can identify the expected value for this KPI-Restriction. This process can be seen in Algorithm 16, where we iterate over all the KPIs defined in the strategic model. For each one of them, we identify every possible dimension-instance pair set and we calculate the value of the KPI for it. We then find all the parent market segments of this sub-dimension, and calculate the ratio of its value to the value of the KPI for the parent market segment. Then using an user defined function, we combine all these ratios in order to choose either the most meaningful or an aggregation of them.

An example for the employment KPI is the following. Starting with the aggregation of employment all over Europe, we calculate the employment for all countries, then for

all regions and finally for the eurozone dimension. At each step we compare this KPI to the KPI of its parent market segments. For example, we compare Spain to Europe, Spain to Eurozone, Spain to Southern Europe. We then use the user defined function to choose the most appropriate ratio, or to aggregate them, thus adjusting the global target value to this sub-dimension. In the same way we compare Eurozone to Europe, each Region to Europe, and adjust the target values for the related sub-dimensions.

**Example 5 (Temporary Reforms: Automatic Projection on the Time dimension)** *As per our previous example, we could take a look at similar situations, e.g., previous Temporary Reforms, and mine the percentages over the different years with regards to the overall target value.*

**Example 6 (Development Level: Automatic Projection on the Space dimension)** *We could automatically calculate the average unemployment rates for the past years for all the countries in East EU, and compare each country's average to the total average. This ratio can then be used to adjust the expectations for each country on its own.*

#### Partial History Selection.

Further on, in our scenario we only want to select parts of the data warehouse (from 2000 onwards) for performing the target value projections, instead of the complete historic knowledge. Partial history selection can become more complex, selecting isolated parts of the history. An example could be an economic crisis, where we want to get unemployment data only from periods where there was a financial crisis affecting some parts of Europe, and use them to adjust the target values for each sub-market. In order to deal with these situations, we can have a Knowledge Base that contains historic situations pointing to parts of the data warehouse that contain data related to them. Thus, we retrieve only the values from the relevant time intervals.

Summarizing, before proceeding to the monitoring step, we need to know how to fill the missing and target values of the KPIs, both for the aggregated data and for each sub-market. As we have shown, we can project segmentations using the previous data, either by:

- Using the current information available about the evolution of each KPI.
- Performing Partial History Selection for focusing on specific situations that affect this KPI that have occurred in the past, using a pre-annotated knowledge-base of past situations and types of situations.

**Algorithm 12:** Performance Diagnosis Setup

**Data**: KPIs: $\mathcal{K}$, Predefined Target Value Projections: $P$, Data Warehouse: $DW$
**Result**: $\mathcal{K}_{proj}$

```
1  K_proj = K;
2  foreach k ∈ K do
3      foreach Possible Sub-Dimensions d of k do
4          k_restricted = SelectSubDimension(k, d);
5          if hasPredefinedTargetValues(k_restricted, P) then
6              k_restricted.targetValues = getTargetValues(k,d,P);
7          else
8              restrVal = calculateKPI(k, d, DW);
9              p_all = findParentMarketDimensions(k_restricted);
10             parentKPIValueRatios = [];
11             i = 0;
12             foreach p ∈ p_all do
13                 parVal = calculateKPI(k, p, DW);
14                 parentKPIValueRatios[i++] = restrVal / parVal ;
15             k_restricted.targetValues = calculateTargetValues(parentKPIValueRatios);
16         append(k_restricted, K_proj);
```

**Expectation Diagnostics.**

In the previous sections we have focused on preparing the necessary data to answer **Performance Diagnostics** questions such as "Will we meet our goal for Employment Rate by year 2020?". However, as introduced in Section 7.2, we are also interested in being able to identify if there are hidden anomalies such as countries deviating from their usual behavior. For example, it is relevant to know if Spain or Greece are deviating from the employment behavior of other countries and when they started deviating. These questions fall into the **Expectation Diagnostics** category.

In order to support **Expectation Diagnostics**, we need to perform clustering at the base level of our data warehouse, such that we group our data and introduce hidden dimensions that can be used for aggregating their members, and producing trends that describe them. These aggregated trends, can be used to monitor for parts of the warehouse that are deviating from their previous clusters.

Expectation Diagnostics is the process that allows an automated monitoring system to compare current KPI trends to expected KPI trends. Expectation differs from

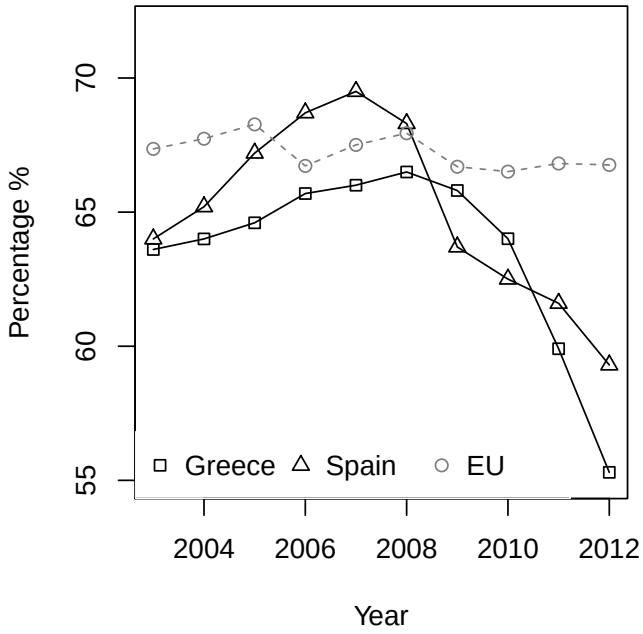



Figure 7.6: Cluster of Spain and Greece, over EU employment percentage

Performance in the sense that it tries to capture differences among different layers of aggregation and different parts of the data warehouse, where the same trends should traditionally be observed. An example is that sub-markets should most of the time follow the same trends as their parent market segments. For example, unemployment in Spain could be expected to follow the trend of unemployment of Europe. If this is not the case, then Spain constitutes a special case that is worth being reported to an analyst. At the same time, this might not always be the case, as there might be sub-clusters within a certain hierarchy level, that do not necessarily follow the same trends [200]. For our scenario, we can see the trends and clusters of Spain and Greece compared with the trend of EU in Figure 7.6.

In order to be able to capture such insights, we need to support data series similarity queries for performing the following actions.

1. Clustering data series from different parts of the data warehouse, thus producing more meaningful levels of aggregation, where the aggregations follow the same trends as their components. An example would be the creation of clusters of countries that follow the same unemployment trends, when this clustering is not provided by any of the current dimensions (e.g. Region, Eurozone, etc.), but rather from a hidden dimension that could be introduced by a clustering algo-

rithm.

2. Comparing data series with their parent market segments, either computed via clustering, or physically located in the Dimensions Schema, in order to identify unexpected trends.

Consequently, similarity queries have to be used both while monitoring for unexpected deviations, and while periodically trying to find or update existing clusters in the data warehouse. As a result, such systems should be able to efficiently answer similarity queries on large collections of data series defined as follows.

**Definition 14 (KPI Data-Series Similarity Query)** *Given a two KPI Data Series* $ds_1 = DS_{k_1}^{d_1}(p_{1_s}, p_{1_e}, p_{1_{in}})$ *and* $ds_2 = DS_{k_2}^{d_2}(p_{2_s}, p_{2_e}, p_{2_{in}})$ *of equal size, for two KPIs* $k_1$ *and* $k_2$*, we are interested in finding out the distance between* $ds_1$ *and* $ds_2$*, by using a distance function* $\delta$ *that returns a real number representing this distance.*

$$series\_similarity(ds_1, ds_2, \delta) \text{ returns } x \in \mathbb{R}$$

The problem of answering similarity queries in databases of data series was first introduced by [11] in 1993. As we have seen in the previous chapters, a common approach to handle such queries is by reducing the dimensionality of the data using a dimensionality reduction technique [38, 93] and then building a specialized index structure [13, 166, 30]. Some example distance functions that can be used on top of such indices are the Euclidean Distance, Dynamic Time Warping [151] and others.

After the setup step, we should end up with a set of KPIs and their corresponding target value projections for each different sub-part of each KPI. Subsequently these new Sub-KPIs will be used for Performance Diagnosis both at a high level as well as at a lower level, thus allowing the analyst to adjust a strategy to all the important parts of the data that demonstrate a bad performance. Moreover, for Expectation Analysis we should end up with a new set of dimensions inserted in the Dimensions Schema, such that we are able to use them as if they were preexisting in the data warehouse.

### 7.3.3 Monitoring Step

The last step in the process is the monitoring step. The monitoring step is responsible for monitoring all the KPIs related to each goal, aggregate their statuses, and provide insights for each one of them. The first step of the process is the one related to **Performance Diagnosis**. Here, the system should be able to identify whether a goal is failing

**Algorithm 13:** Performance Diagnosis Monitoring

**Data**: Goals: $G$, Data Warehouse: $DW$, Time Dimension: $t$, forecast step: f, forecast prediction model: p
**Result**: statuses

```
statuses = [];
foreach g ∈ 𝒢 do
    foreach Possible Sub-dimensions d, except time do
        goalKPIStatuses = [];
        foreach k ∈ getKPIs(g) do
            k_restricted = SelectSubDimension(k, d);
            k_data_series = DataSeriesGrouping(k, t);
            ds = ComputeDataSeries(k_data_series);
            futureVal = forecast(ds, p, f);
            goalKPIStatuses.add(getKPIStatus(k_restricted, futureVal));
        statuses.add(aggregateKPIStatuses(kpiStatuses, g));
```

or not. In our scenario, we need to know if we are currently meeting or not our target for Employment and Education.

- If the overall goal is failing, or about to fail, we need to drill down in our data warehouse, by restricting the KPIs to various dimensions, in search of the sub-markets responsible for the overall failure.
- If the overall goal is succeeding, we need to point out to the analyst the parts of the Warehouse with the greatest success, that are probably responsible for this good status, as well as the ones that can still be improved.

A baseline algorithm would start with the general, unrestricted KPIs, and gradually produce restricted KPIs for all dimension-instance pairs. Subsequently, by calculating their values it should be able to produce analytics for each sub-market. Moreover, by performing Data-Series Grouping operations for each one of the restricted and unrestricted KPIs, forecasting algorithms can be used for performing this kind of analysis ahead of time. The output of the algorithm should be an overall status for the general goal and various insights for the status of this goal for various sub-markets. This algorithm can be seen in Algorithm 11, where we start by iterating over all the goals, for each goal we iterate over all the related KPIs and calculate every possible dimension restriction in all dimension-instance pairs. For each one of them we calculate the KPI

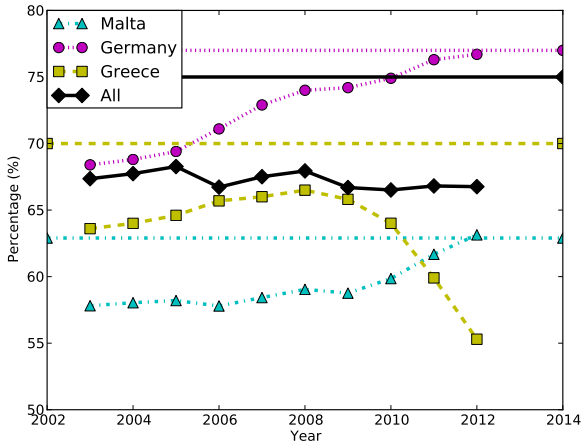



Figure 7.7: EU failing to meet its target, but Malta succeeds, Germany is about to succeed, and Greece will probably fail.

Data Series on the time dimension, until the current time point. We then try to forecast the value for a future time $f$, using a prediction model $p$, both of which we have as input from the analyst. This future value (if f is set to 0, corresponds to now), is translated to a status. By aggregating all these statuses we can calculate the overall status of each KPI-restriction, as well as of the general KPI.

Obviously, the search space can explode really fast, as the number of combinations of dimensions and instances is very large. To overcome this problem various pruning techniques can be used, such as stopping to drill in when a sub-market of the original KPI has been marked as failed. For example, when we identify that East Europe is failing, we could either choose to drill in to the Eurozone dimension, or report this to the analyst and only drill in, on demand.

The last monitoring step is that of Expectation Diagnosis, where we are interested in finding market segments that are outlying with regards to their parent market segments. This process can be done in a top-down way, as described in [105], where we start at the highest level of aggregation and only explore the dimensions that are part of the most dissimilar descendants of this aggregation. This step makes use of the KPI Data Series and clusters previously calculated, and helps the user to find out diverging trends within the data as will be shown in the following section.

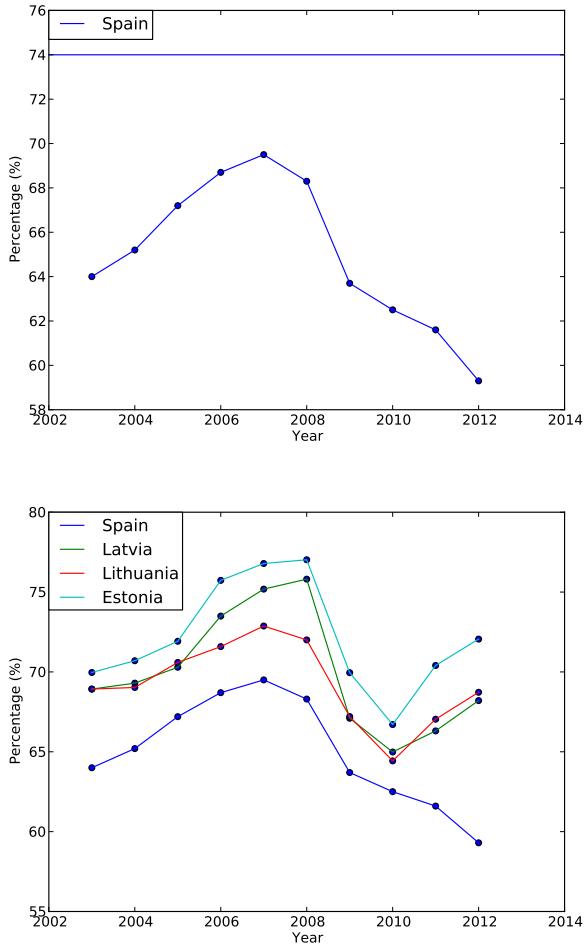



Figure 7.8: Spain failing to succeed as well as it deviates from its previous cluster in 2012

### 7.3.4 Result Analysis

In this section we present the results of following our process for analyzing Employment and Education with regards to Europe 2020 goals. As we can see in Figure 7.7, the **Performance Diagnosis** suggests that we are not expected to meet our goal in Employment if we follow the current trend. However, if analyzed in detail, we can see in Figure 7.7 that some countries present more significant differences with their targets than Europe overall. Malta for example, has already succeeded, while Germany is about to succeed, and Greece will probably fail to do so. In order to discover such insights, Algorithm 16 is run as a setup step to identify the target values for EU and for all the other sub-dimensions in the Data Warehouse. In this case, EU has defined specific targets for each country, and these targets are monitored in a top down fashion

by Algorithm 11.

By means of **Expectation Diagnostics**, we can find out what countries started to diverge from their traditional behavior, and when this started to happen. An example can be seen in Figure 7.8, where we can see that Spain fails to succeed to its target, something that constitutes a Performance Diagnostic. Moreover, while traditionally correlated with the Baltic countries, it starts to deviate and this is an Expectation Diagnostic, as the last years' course is not expected considering previous clusterings.

Finally, with regards to Education, Spain had been deviating from its target until the beginning of the economic crisis, when, unexpectedly, it changed its trend.

As we can see, with our process, we can obtain **Performance and Expectation Diagnostics** that provide important information in order to analyze in-depth the performance of the business. Furthermore, these results can be reflected into the strategic model in order to analyze their impact. It is worth noting however, that finding the reasons for latent anomalies requires additional considerations and is out of the scope of this paper.

## 7.4 Summary

Monitoring the business requires an in-depth analysis of the Key Performance Indicators (KPIs), as otherwise, problems within the sub-markets will go unnoticed until they threaten the global target. However, given the complexity of the search space it can become a daunting task if performed manually. In this chapter we presented a semi-automatic approach to tackle this problem by (i) modeling the business strategy and deciding on what KPIs should be monitored, (ii) describing a process to analyze sub-markets and evaluate their performance, and (iii) specifying a set of algorithms to perform this process. Furthermore, we tested our approach by means of a real case study on the Europe 2020 framework, publicly available. The benefits of our proposal are that we can identify anomalies, relationships, and deviations in the data that are not reflected at an aggregated level. Therefore, we are able to diagnose the existence of anomalies before they become threats for the business.

# Chapter 8

# Conclusions and Future Work

## 8.1 Conclusions

In this thesis, we have discussed the importance of efficient similarity search for a large set of data mining operations. This is the case both for raw data series, as well as for data series that are enriched with semantic meta-data, aimed at supporting business intelligence applications. We have shown that full state-of-the-art indexes cannot cope with the data deluge, often requiring multiple days before allowing the analysts to start answering queries. As a result, hindering the interactivity of exploration systems. Through our experiments, we have demonstrated that adaptive indexing can help scientists cope with the vast amounts of data that they have to process. Such techniques are able to allow scientists to perform similarity search at interactive speeds. Our proposed approach, constructs the index incrementally and adaptively, resulting in a very fast initialization process. The index structure then adapts to the query workload, as a result saving time from indexing the complete dataset. This time is instead used to create a more refined version of the index, for the parts of the data that are related to the user's queries.

Using this data structure we are able to answer approximate queries in multi-terrabyte datasets in mere milliseconds, while we have also presented algorithms for efficient exact query answering. Our exact search algorithm facilitates sequential disk scans in order to speed up its operation. Further on, in the case where analysts are still interested in indexing the complete dataset, we have also developed a full index construction method based on ADS, which outperforms the state-of-the-art, by performing a double pass over the raw dataset, instead of costly random I/O operations.

In order to demonstrate all these ideas, we have also developed RINSE, an inter-

active data series exploration platform. This system allows the users to experience how adaptive indexing works, and the quick access it provides to the data. It is also a testbed for the creation of new data exploration user interfaces, as it allows users to graphically explore data series using their mouse or touch screen.

In addition to speeding up similarity search, we also focused on the problem of how to to systematically characterize a data series query workload, and subsequently, how to generate queries with desired properties. This is an important step, that has never been studied in the past in the context of data series indexes. The output of this work was a query workload generator, which generates queries of predefined hardness, given by the user as input.

Finally, we motivated the need for a system that is able to continuously monitor a data warehouse based on queries generated from the Strategic Model of an organization and discussed the importance of data series similarity in such a context. Our proposed systems should be able to efficiently identify trends in regards to these prespecified objectives, and also to monitor the warehouse for expected or unexpected threats and opportunities in the data as well as their causes.

## 8.2 Future Work

Our future work can be organized in 4 distinct axes. These include the field of distributed adaptive indexing for data series, the development of a generic data series storage and retrieval systems, the creation of novel interactive data exploration techniques, and finally the enhancement of business intelligence systems with efficient data series queries processing. We elaborate on each one of these axes in the following subsections.

### 8.2.1 Distributed Adaptive Indexing

Adaptive data series exploration opens numerous future research opportunities. One of the most challenging paths is the application of our initial ideas in massively distributed settings, which would allow the exploitation of cloud resources and enable multiple scientists to co-explore massive data collections. While our current implementation is limited to a single node scenario, ADS can be naturally parallelized, e.g., by distributing different sub-trees to different nodes of a cluster system [178, 184, 56]. Moreover, in order to ensure uniform utilization of the complete infrastructure, each node can host more than one sub-tree, including both hot and cold parts of the index.

### 8.2.2 Data Series Management Systems

Massive data series collections are becoming a reality for virtually every scientific and social domain. Systems like Relational and Array Databases are not a suitable solution, since none of these systems offers native support for data series. Our vision is to design and develop general-purpose Data Series Management Systems [122, 123], able to cope with big data series, that is, very large and fast-changing collections of data series, which can be heterogeneous (i.e., originate from disparate domains and thus exhibit very different characteristics), and which can have uncertainty in their values (e.g., due to inherent errors in the measurements). Just like databases abstracted the relational data management problem and offered a black box solution that is now omnipresent, we propose that Data Series Management Systems should rely on declarative processing to allow analysts that are not experts in data series management, as well as common users, to tap in the goldmine of the massive and ever-growing data series collections they (already) have. Additionally, one of our goals is to also integrate ADS+, as well as other data series indexes and summarizations into a general data series management system [122]. This will optimize similarity search, and support interactive exploration of big data series.

#### Data Series Storage Layer

The first step towards this direction is the development of novel data series storage systems. While data series indexes have been proposed in the context of similarity search for speeding up data mining algorithms, we claim that data series indexes can be used to speed up other queries as well. This is the case for simple range queries, which retrieve a subset of the positions for some of the data series. Such queries very frequently refer only to a small subset of the data series collection, and in such cases, query processing would be substantially improved by intelligently reorganizing data accordingly. For example, by grouping together data series commonly accessed by the same queries, or that share statistical properties. Additionally, data series could be horizontally partitioned (over the positions axis) in a way that reflects the selectivity (in number of points) of the queries of the current workload. Along these lines, our long term vision is to develop a specialized data series storage layer that goes beyond relational and array databases, by being able to both efficiently re-organize data series in optimal groups, but also partition each group in a way that is optimal for the query workload.

### 8.2.3 Interactive Data Exploration

In regards to data exploration, there are various research topics, which cut across the fields of interactive data exploration, human computer interaction, and data management. Effectively browsing through large collections of data series in a visual way, efficiently isolating the data of interest, is the holy grail of interactive data exploration techniques. Our adaptive data series indexes can form the core of such products, allowing analytics at speed-of-thought speeds.

### 8.2.4 Enhancing Business Intelligence Systems

Finally, business intelligence applications can also be enhanced by the introduction of adaptive data series indexes. This will allow such tools to efficiently identify clusters and outliers. Such applications, require scalable and efficient data structure to handle the query workload. Further on the queries generated are targeted to the parts of the data, which are related to the strategic goals of a company. As a result there are various data layout choice (or index building) decisions that could be guided by analyzing these goals. In regards to the enhancement of such tools, our future work is additionally focused on identifying the potential causes and solutions of such outliers. Finally, another interesting research path is that of identifying hidden relationships in the strategic model through the use of clustering.

# Bibliography


[1] FAME – Time Series Data Management Solutions. https://fame.sungard.com/ourCompany/usrcon/usrcon99/ucpapers/fme_fred.pdf.

[2] FLibreTexts.org – The Mass Spectra of Elements. http://chem.libretexts.org/Core/Analytical_Chemistry/Instrumental_Analysis/Mass_Spectrometry/The_Mass_Spectra_of_Elements.

[3] ONETICK – Enabling the Success of Quantitative Research and Trading. https://www.onetick.com/images/pdf/OneTick-EnablingSuccessofQuantResearchandTradingv2.pdf.

[4] The cybertory project. http://cybertory.org/exercises/seq1_shotgun/, 2009.

[5] Adhd-200. http://fcon_1000.projects.nitrc.org/indi/ adhd200/, 2011.

[6] QualiMaster A configurable real-time Data Processing Infrastructure mastering autonomous Quality Adaptation – Deliverable D1.1: Initial Use Cases and Requirements. Technical report, QualiMaster Project, 2014.

[7] Sloan digital sky survey. https://www.sdss3.org/dr10/ data_access/volume.php, 2015.

[8] Incorporated Research Institutions for Seismology – Seismic Data Access. http://ds.iris.edu/data/access/, 2016.

[9] D. Achakeev and B. Seeger. Efficient bulk updates on multiversion b-trees. *PVLDB*, 6(14):1834–1845, 2013.

[10] A+Dev. A+: History of A+. http://www.aplusdev.org/About/index.html.

[11] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. In *FODO*, 1993.

[12] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: the stanford stream data manager. In *SIGMOD*, 2003.

[13] I. Assent, R. Krieger, F. Afschari, and T. Seidl. The ts-tree: Efficient time series search and retrieval. In *EDBT*, 2008.

[14] D. Barone, L. Jiang, D. Amyot, and J. Mylopoulos. Composite indicators for business intelligence. *Conceptual Modeling–ER 2011*, 2011.

[15] D. Barone, T. Topaloglou, and J. Mylopoulos. Business intelligence modeling in action: a hospital case study. In *Advanced Information Systems Engineering*, pages 502–517. Springer, 2012.

[16] P. Baumann, P. Furtado, R. Ritsch, and N. Widmann. Geo/environmental and medical data management in the rasdaman system. In *VLDB*, 1997.

[17] S. D. Bay, D. Kibler, M. J. Pazzani, and P. Smyth. The uci kdd archive of large data sets for data mining research and experimentation. In *SIGKDD Explorations*, 2000.

[18] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

[19] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *VLDB*, 1996.

[20] M. R. Berthold and F. Höppner. On clustering time series using euclidean distance and pearson correlation. *CoRR*, abs/1601.02213, 2016.

[21] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *ICDT*, 1999.

[22] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Mobile Data Management, Second International Conference, MDM 2001, Hong Kong, China, January 8-10, 2001, Proceedings*, pages 3–14, 2001.

[23] G. E. Box, G. M. Jenkins, and G. C. Reinsel. *Time series analysis: forecasting and control*, volume 734. Wiley, 2011.

[24] D. R. Brillinger. Time series: general. *Int. Encyc. Social and Behavioral Sciences*.

[25] R. Bryant. Data-Intensive Scalable Computing for Scientific Applications. *Computing in Science & Engineering*, 13(6):25–33, 2011.

[26] Y. Bu, T. wing Leung, A. W. chee Fu, E. Keogh, J. Pei, and S. Meshkin. Wat: Finding top-k discords in time series database. In *SDM*, 2007.

[27] Y. D. Cai, D. Clutter, G. Pape, J. Han, M. Welge, and L. Auvil. MAIDS: mining alarming incidents from data streams. In *SIGMOD*, 2004.

[28] V. R. B. G. Caldiera and H. D. Rombach. The goal question metric approach. *Encyclopedia of software engineering*, 2(1994):528–532, 1994.

[29] S. H. Cameron. Piece-wise linear approximations. Technical report, DTIC Document, 1966.

[30] A. Camerra, T. Palpanas, J. Shieh, and E. Keogh. iSAX 2.0: Indexing and mining one billion time series. In *ICDM*, 2010.

[31] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. Keogh. Beyond One Billion Time Series: Indexing and Mining Very Large Time Series Collections with iSAX2+. *KAIS*, 39(1):123–151, 2014.

[32] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *SIGMOD*, 1994.

[33] D. Carney, U. Etintemel, N. Tatbul, M. Stonebraker, D. J. Abadi, S. Zdonik, C. Convey, S. Lee, and M. Cherniack. Aurora: a new model and architecture for data stream management. *The VLDB Journal The International Journal on Very Large Data Bases*, 12:120–139, 2003.

[34] A. M. Castillejos. *Management of time series data*. PhD thesis, University of Canberra, 2006.

[35] U. Cetintemel and D. Abadi. The Aurora and Borealis Stream Processing Engines. *Data Stream Management: Processing High-Speed Data Streams, Springer-Verlag*, 2006.

[36] K. Chakrabarti, E. J. Keogh, S. Mehrotra, and M. J. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM Trans. Database Syst.*, 27(2):188–228, 2002.

[37] P. Chamoni. Temporal structures in data warehousing. In *DEXA*, 1999.

[38] K.-P. Chan and A.-C. Fu. Efficient time series matching by wavelets. In *ICDE*, 1999.

[39] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: a survey. *ACM Computing Surveys*, 41(3):1–58, July 2009.

[40] R. Chandra and A. Segev. Managing temporal financial data in an extensible database. In *VLDB*, 1993.

[41] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[42] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, Mar. 1997.

[43] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD*, 2005.

[44] Q. Chen, L. Chen, X. Lian, Y. Liu, and J. X. Yu. Indexable pla for efficient similarity search. In *VLDB*, 2007.

[45] Y. Chen, G. Dong, J. Han, B. Wah, and J. Wang. Multi-dimensional regression analysis of time-series data streams. In *VLDB*, 2002.

[46] D. Chimenti, A. B. O'Hare, R. Krishnamurthy, S. Tsur, C. West, and C. Zaniolo. An overview of the LDL system. *IEEE Data Eng. Bull.*, 10(4):52–62, 1987.

[47] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, 1997.

[48] E. Codd, S. Codd, and C. Salley. *Providing OLAP (on-line Analytical Processing) to User-analysts: An IT Mandate*, volume 32. Codd & Associates, 1993.

[49] P. Cudré-Mauroux, H. Kimura, K. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. J. DeWitt, B. Heath, D. Maier, S. Madden, J. M. Patel, M. Stonebraker, and S. B. Zdonik. A demonstration of scidb: A science-oriented DBMS. *PVLDB*, 2(2):1534–1537, 2009.

[50] M. Dallachiesa, B. Nushi, K. Mirylenka, and T. Palpanas. Uncertain time-series similarity: Return to the basics. *PVLDB*, 5(11):1662–1673, 2012.

[51] M. Dallachiesa, T. Palpanas, and I. F. Ilyas. Top-k nearest neighbor search in uncertain data series. *PVLDB*, 8(1):13–24, 2014.

[52] A. Dardenne, A. Van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of computer programming*, 20(1), 1993.

[53] W. Dreyer, A. K. Dittrich, and D. Schmidt. Research perspectives for time series management systems. *ACM SIGMOD Record*, 23(1):10–15, 1994.

[54] W. Dreyer, A. K. Dittrich, and D. Schmidt. Using the calanda time series management system. In *ACM SIGMOD Record*, volume 24, page 489. ACM, 1995.

[55] Druid. Druid — Real-time Exploratory Analytics on Large Datasets.

[56] C. du Mouza, W. Litwin, and P. Rigaux. SD-Rtree: A scalable distributed rtree. In *ICDE*, 2007.

[57] W. W. Eckerson. *Performance dashboards: measuring, monitoring, and managing your business*. Wiley, 2010.

[58] P. Esling and C. Agón. Time-series data mining. *ACM Comput. Surv.*, 45(1):12, 2012.

[59] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, 1994.

[60] H. G. Funkhouser. A note on a tenth century graph. *Osiris*, 1:260–262, 1936.

[61] T. M. Ghanem, R. Shah, M. F. Mokbel, W. G. Aref, and J. S. Vitter. Bulk operations for space-partitioning trees. In *ICDE*, 2004.

[62] M. Golfarelli. A survey on temporal data warehousing. *International Journal of Data Warehousing*, 5, 2009.

[63] G. Graefe, F. Halim, S. Idreos, H. A. Kuno, and S. Manegold. Concurrency control for adaptive indexing. *PVLDB*, 5(7):656–667, 2012.

[64] G. Graefe, F. Halim, S. Idreos, H. A. Kuno, S. Manegold, and B. Seeger. Transactional support for adaptive indexing. *VLDB J.*, 23(2):303–328, 2014.

[65] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1), 1997.

[66] T. B. R. Group. The business motivation model.

[67] C. Gupta, S. Wang, I. Ari, and M. Hao. Chaos: A data stream analysis architecture for enterprise applications. *IEEE Conference on Commerce and Enterprise Computing*, 2009.

[68] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[69] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic Database Cracking: Towards Robust Adaptive. *PVLDB*, 5(6):502–513, 2012.

[70] R. Hall. *Simple techniques for constructing explanatory models of complex systems for policy analysis*. Faculty of Administrative Studies, University of Manitoba, 1978.

[71] M. a. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, a. C. Catlin, a. K. Elmagarmid, M. Eltabakh, M. G. Elfeky, T. M. Ghanem, R. Gwadera, I. F. Ilyas, M. Marzouk, and X. Xiong. Nile: A query processing engine for data streams. In *ICDE*, 2004.

[72] J. Han, Y. Chen, G. Dong, J. Pei, B. W. Wah, J. Wang, and Y. D. Cai. Stream Cube: An Architecture for Multi-Dimensional Analysis of Data Streams. *Distributed and Parallel Databases*, 18(2):173–197, 2005.

[73] H. Hassanieh, P. Indyk, D. Katabi, and E. Price. Simple and practical algorithm for sparse fourier transform. In *SODA*, 2012.

[74] J. Horkoff, D. Barone, L. Jiang, E. Yu, D. Amyot, A. Borgida, and J. Mylopoulos. Strategic business modeling: representation and reasoning. *Software & Systems Modeling*, pages 1–27, 2012.

[75] P. Huijse, P. A. Estévez, P. Protopapas, J. C. Principe, and P. Zegers. Computational intelligence challenges and applications on large-scale astronomical time series databases. *IEEE Comp. Int. Mag.*, 9(3), 2014.

[76] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results ? In *CIDR*, 2011.

[77] S. Idreos, M. L. Kersten, and S. Manegold. Updating a Cracked Database. In *SIGMOD*, 2007.

[78] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *CIDR*, 2007.

[79] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing Tuple Reconstruction in Column-stores. In *SIGMOD*, 2009.

[80] S. Idreos and E. Liarou. dbtouch: Analytics at your fingertips. In *CIDR*, 2013.

[81] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *PVLDB*, 4(9):585–597, 2011.

[82] InfluxDB. InfluxDB - Open Source Time Series, Metrics, and Analytics Database (http://influxdb.com/).

[83] M. Jarke, M. Lenzerini, Y. Vassiliou, and P. Vassiliadis. *Fundamentals of Data Warehouses*. Springer-Verlag, 2003.

[84] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *TPAMI*, 33(1):117–128, 2011.

[85] L. Jiang, D. Barone, and D. Amyot. Strategic models for business intelligence. *Conceptual Modeling – ER 2011*, 2011.

[86] R. S. Kaplan and D. P. Norton. *Strategy maps: Converting intangible assets into tangible outcomes*. Harvard Business Press, 2004.

[87] R. S. Kaplan, D. P. Norton, R. Dorf, and M. Raitanen. *The balanced scorecard: translating strategy into action*, volume 4. Harvard Business school press Boston, 1996.

[88] A. Karakasidis, P. Vassiliadis, and E. Pitoura. ETL queues for active data warehousing. In *IQIS*, 2005.

[89] K. Kashino, G. Smith, and H. Murase. Time-series active search for quick retrieval of audio and video. In *ICASSP*, 1999.

[90] S. Kashyap and P. Karras. Scalable knn search on vertically stored time series. In *KDD*, 2011.

[91] E. Keogh and M. Pazzani. Scaling up dynamic time warping to massive datasets. In *PKDD*, 1999.

[92] E. J. Keogh. Fast similarity search in the presence of longitudinal scaling in time series databases. *Tools with Artificial Intelligence*, 1997.

[93] E. J. Keogh, K. Chakrabarti, M. J. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and Information Systems*, 3(3):263–286, 2001.

[94] E. J. Keogh, J. Lin, S.-H. Lee, and H. V. Herle. Finding the most unusual time series subsequence: algorithms and applications. *Knowledge and Information Systems*, 11(1):1–27, 2007.

[95] E. J. Keogh and M. J. Pazzani. An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback. In *KDD*, 1998.

[96] R. Kimball and M. Ross. *The data warehouse toolkit: the complete guide to dimensional modeling*. Wiley, 2011.

[97] J. Knight, D. Heimbigner, A. L. Wolf, E. L. Wolf, A. Carzaniga, A. Carzaniga, J. Hill, J. Hill, P. Devanbu, P. Devanbu, M. Gertz, and M. Gertz. The willow architecture: Comprehensive survivability for large-scale distributed applications. In *Distributed Applications.", Intrusion Tolerance Workshop, Dependable Systems and Networks (DSN 2002), Washington DC*, 2001.

[98] F. Korn, H. V. Jagadish, and C. Faloutsos. Efficiently supporting ad hoc queries in large datasets of time sequences. In *SIGMOD*, 1997.

[99] H. Kremer, S. Günnemann, A.-M. Ivanescu, I. Assent, and T. Seidl. Efficient processing of multiple dtw queries in time series databases. In *SSDBM*, 2011.

[100] R. Lamb. *Competitive strategic management*. Prentice-Hall, Englewood Cliffs, NJ, 1984.

[101] A. Lerner and D. Shasha. Aquery: Query language for ordered data, optimization techniques, and experiments. In *VLDB*, 2003.

[102] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady*, volume 10, page 707, 1966.

[103] C.-S. Li, P. Yu, and V. Castelli. Hierarchyscan: a hierarchical similarity search algorithm for databases of long sequences. In *ICDE*, 1996.

[104] Q. Li and G. D. Clifford. Dynamic time warping and machine learning for signal quality assessment of pulsatile signals. *Physiological Measurement*, 33(9):1491, 2012.

[105] X. Li and J. Han. Mining approximate top-k subspace anomalies in multi-dimensional time-series data. In *VLDB*, 2007.

[106] J. Lin, E. Keogh, and S. Lonardi. A symbolic representation of time series, with implications for streaming algorithms. *Research issues in data mining and knowledge discovery*, page 2, 2003.

[107] J. Lin, E. Keogh, and W. Truppel. Clustering of streaming time series is meaningless. In *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pages 56–65. ACM, 2003.

[108] Logical Information Machines. MIM Data and Development Guide. 9646, 2008.

[109] J. Macqueen. Some methods for classification and analysis of multivariate observations. In *In 5-th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[110] A. Maté, J. Trujillo, and J. Mylopoulos. Conceptualizing and specifying key performance indicators in business strategy models. In *CASCON*. 2012.

[111] A. Maté, K. Zoumpatianos, T. Palpanas, J. Trujillo, J. Mylopoulos, and E. Koci. A systematic approach for dynamic targeted monitoring of kpis. In *CASCON*, 2014.

[112] A. Mendelzon and A. Vaisman. Temporal Queries in OLAP. In *VLDB*, 2000.

[113] M. Middelfart. Improving business intelligence speed and quality through the ooda concept. In *DOLAP*, 2007.

[114] M. Middelfart and T. Pedersen. Implementing sentinels in the targit bi suite. In *ICDE*, 2011.

[115] S. Monna, G. Falcone, L. Beranzoli, F. Chierici, G. Cianchini, M. De Caro, A. De Santis, D. Embriaco, F. Frugoni, G. Marinaro, et al. Underwater geophysical monitoring for european multidisciplinary seafloor and water column observatories. *Journal of Marine Systems*, 130:12–30, 2014.

[116] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[117] A. Mueen, E. J. Keogh, and N. B. Shamlo. Finding time series motifs in disk-resident data. In *ICDM*, 2009.

[118] A. Mueen, E. J. Keogh, Q. Zhu, S. Cash, and M. B. Westover. Exact discovery of time series motifs. In *SDM*, 2009.

[119] R. Nag and D. Hambrick. What is strategic management, really? Inductive derivation of a consensus definition of the field. *Strategic Management*, 955, 2007.

[120] Newts. Newts (http://opennms.github.io/newts/).

[121] OpenTSDB. OpenTSDB - A Distributed, Scalable Monitoring System (http://opentsdb.net/).

[122] T. Palpanas. Data Series Management. *ACM SIGMOD Record*, 44(2):47–52, aug 2015.

[123] T. Palpanas. Big sequence management: A glimpse of the past, the present, and the future. In *SOFSEM 2016: Theory and Practice of Computer Science - 42nd International Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 23-28, 2016, Proceedings*, pages 63–80, 2016.

[124] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental maintenance for non-distributive aggregate functions. *VLDB*, 2002.

[125] T. Palpanas, M. Vlachos, E. J. Keogh, and D. Gunopulos. Streaming time series summarization using user-defined amnesic functions. *TKDE*, 20(7):992–1006, 2008.

[126] T. Palpanas, M. Vlachos, E. J. Keogh, D. Gunopulos, and W. Truppel. Online amnesic approximation of streaming time series. In *ICDE*, pages 339–349, 2004.

[127] J. Paparrizos and L. Gravano. k-shape: Efficient and accurate clustering of time series. In *SIGMOD*, 2015.

[128] D. Parmenter. *Key performance indicators (KPI): developing, implementing, and using winning KPIs*. Wiley, 2010.

[129] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. Gorilla: a fast, scalable, in-memory time series database. In *VLDB*, 2015.

[130] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N. Frantzell. Supporting Streaming Updates in an Active Data Warehouse. In *ICDE*, 2007.

[131] D. Rafiei and A. Mendelzon. Similarity-based queries for time series data. In *SIGMOD*, 1997.

[132] D. Rafiei and A. Mendelzon. Efficient retrieval of similar time sequences using dft. In *ICDE*, 1998.

[133] Raghu Ramakrishnan, Michael Cheng, Miron Livny and P. Seshadri. What's next? Sequence queries, 1994.

[134] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *KDD*, 2012.

[135] T. Rakthanmanon, E. J. Keogh, S. Lonardi, and S. Evans. Time Series Epenthesis: Clustering Time Series Streams Requires Ignoring Some Data. *ICDE*, 2011.

[136] C. A. Ralanamahatana, J. Lin, D. Gunopulos, E. Keogh, M. Vlachos, and G. Das. Mining time series data. In *Data mining and knowledge discovery handbook*, pages 1069–1103. Springer, 2005.

[137] R. Ramakrishnan, W. G. Roth, P. Seshadri, D. Srivastava, and S. Sudarshan. The CORAL deductive database system. In *SIGMOD*, 1993.

[138] R. Ramakrsihnan, D. Donjerkovic, A. Ranganathan, K. Beyer, and M. Krishnaprasad. SRQL: Sorted Relational Query Language. In *SSDBM*, 1998.

[139] K. V. Ravi Kanth, D. Agrawal, and A. Singh. Dimensionality reduction for similarity searching in dynamic databases. In *SIGMOD*, 1998.

[140] U. Raza, A. Camerra, A. L. Murphy, T. Palpanas, and G. P. Picco. Practical data prediction for real-world wireless sensor networks. *IEEE Trans. Knowl. Data Eng.*, 27(8):2231–2244, 2015.

[141] Red Brick Systems. Star Schemas and StarJoin Technology. 1995.

[142] Red Brick Systems. Decision-Makers, Business Data, and RISQL. 1996.

[143] Red Brick Systems. Star schema processing for complex queries. 1997.

[144] S. Richter, J.-A. Quiane-Ruiz, S. Schuh, and J. Dittrich. Towards zero-overhead static and adaptive indexing in hadoop. *VLDBJ*, 23(3):469–494, 2014.

[145] P. Rodrigues, J. Gama, and J. Pedroso. Hierarchical clustering of time-series data streams. *TKDE*, 20(5):615–627, 2008.

[146] S. Rogers. Big data is scaling bi and analytics, 1 Sep 2011.

[147] J. Roos, Leslie L. and R. I. Hall. Influence diagrams and organizational power. *Administrative Science Quarterly*, 25(1).

[148] W. G. Roth, R. Ramakrishnan, and P. Seshadri. MIMSY: A system for analyzing time series data in the stock market domain. In *Proceedings of the Workshop on Programming with Logic Databases. In Conjunction with ILPS, Vancouver, B.C., October 30, 1993*, pages 33–43, 1993.

[149] R. Sadri, C. Zaniolo, and A. M. Z. andJafar Adibi. A Sequential Pattern Query Language for Supporting Instant Data Minining for e-Services. In *VLDB*, 2001.

[150] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Optimization of sequence queries in database systems. *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems - PODS '01*, pages 71–81, 2001.

[151] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 26(1):43–49, 1978.

[152] S. Sarawagi. Explaining differences in multidimensional aggregates. In *VLDB*, 1999.

[153] S. Sarawagi. User-adaptive exploration of multidimensional data. In *VLDB*, 2000.

[154] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-Driven Exploration of OLAP Data Cubes. In *EDBT*, 1998.

[155] D. Sasha. Kdb+ Database and Language Primer, 2005.

[156] P. Schäfer and M. Högqvist. Sfa: A symbolic fourier approximation and index for similarity search in high dimensional datasets. In *EDBT*, 2012.

[157] D. Schmidt, A. K. Dittrich, W. Dreyer, and R. Marti. Time series, a neglected issue in temporal database research? In *Recent Advances in Temporal Databases*, pages 214–232. Springer, 1995.

[158] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The Uncracked Pieces in Database Cracking. *PVLDB*, 7(2):97–108, 2013.

[159] P. Seshadri. Generalized Partial Indexes. In *ICDE*, 1995.

[160] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query processing. In *SIGMOD*, 1994.

[161] P. Seshadri, M. Livny, and R. Ramakrishnan. The Design and Implementation of a Sequence Database System. *VLDB*, pages 99–110, 1996.

[162] P. Seshadri and M. Paskin. PREDATOR : An OR-DBMS with Enhanced Data Types. *SIGMOD*, pages 568–571, 1997.

[163] D. Shasha. Tuning time series queries in finance: Case studies and recommendations. *IEEE Data Eng. Bull.*, 22(2), 1999.

[164] D. E. Shasha and Y. Zhu. *High Performance Discovery in Time Series - Techniques and Case Studies*. Monographs in Computer Science. Springer, 2004.

[165] J. Shieh and E. Keogh. iSAX: indexing and mining terabyte sized time series. In *KDD*, 2008.

[166] J. Shieh and E. Keogh. iSAX: disk-aware mining and indexing of massive time series datasets. *Data Mining and Knowledge Discovery*, 19(1), 2009.

[167] V. E. Silva Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos. Awareness requirements for adaptive systems. In *Proceedings of the 6th international symposium on Software engineering for adaptive and self-managing systems*, pages 60–69. ACM, 2011.

[168] S. Soldi, V. Beckmann, W. Baumgartner, G. Ponti, C. R. Shrader, P. Lubiński, H. Krimm, F. Mattana, and J. Tueller. Long-term variability of agn at hard x-rays. *Astronomy & Astrophysics*, 563:A57, 2014.

[169] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson. Big data: astronomical or genomical? *PLoS Biol*, 13(7):e1002195, 2015.

[170] M. Stonebraker. The case for partial indexes. *SIGMOD Record*, 18(4):4–11, 1989.

[171] M. Stonebraker and L. A. Rowe. The design of postgres. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28-30, 1986.*, pages 340–355, 1986.

[172] A. Szalay. Extreme data-intensive scientific computing. *Computing in Science & Engineering*, 13(6):34–41, 2011.

[173] B. Thuraisingham. *Data management systems: Evolution and interoperation*. CRC Press, 1997.

[174] A. Van Lamsweerde, R. Darimont, and P. Massonet. Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. In *RE*, 1995.

[175] P. Vassiliadis, A. Simitsis, and P. Georgantas. A generic and customizable framework for the design of ETL scenarios. *Information Systems*, 2005.

[176] T. K. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics and Systems Analysis*, 4(1):52–57, 1968.

[177] M. Vlachos, D. Gunopulos, and G. Kollios. Discovering similar multidimensional trajectories. In *ICDE*, 2002.

[178] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi. Indexing multi-dimensional data in a cloud system. In *SIGMOD*, 2010.

[179] X. Wang, A. Mueen, H. Ding, G. Trajcevski, P. Scheuermann, and E. Keogh. Experimental comparison of representation methods and distance measures for time series data. *DMKD*, 26(2), 2013.

[180] Y. Wang, P. Wang, J. Pei, W. Wang, and S. Huang. A data-adaptive and dynamic segmentation index for whole matching on time series. In *VLDB*, 2013.

[181] T. Warren Liao. Clustering of time series data - a survey. *Pattern Recognition*, 38(11):1857–1874, Nov. 2005.

[182] R. Xi, N. Lin, and Y. Chen. Compression and aggregation for logistic regression analysis in data cubes. *TKDE*, 21(4):479–492, 2009.

[183] X. Xi, E. J. Keogh, L. Wei, and A. Mafra-Neto. Finding motifs in a database of shapes. SDM, 2007.

[184] Y. Xie, D. Palsetia, G. Trajcevski, A. Agrawal, and A. N. Choudhary. SILVERBACK: scalable association mining for temporal data in columnar probabilistic databases. In *ICDE*, 2014.

[185] D. Yankov, E. Keogh, J. Medina, B. Chiu, and V. Zordan. Detecting time series motifs under uniform scaling. In *KDD*, 2007.

[186] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31(3):9–18, 2002.

[187] L. Ye and E. J. Keogh. Time series shapelets: a new primitive for data mining. In *KDD*, 2009.

[188] B. Yi and C. Faloutsos. Fast Time Sequence Indexing for Arbitrary Lp Norms. In *VLDB*, 2000.

[189] B.-K. Yi, H. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *ICDE*, 1998.

[190] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, volume 93, pages 311–21, 1993.

[191] M. J. Zaki and W. M. Jr. *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press, 2014.

[192] A. Zaslavsky, C. Perera, and D. Georgakopoulos. Sensing as a service and big data. *arXiv preprint arXiv:1301.0159*, 2013.

[193] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *VLDB*, 2003.

[194] J. Zhou and K. A. Ross. Buffering database operations for enhanced instruction cache performance. In *SIGMOD*, 2004.

[195] K. Zoumpatianos, S. Idreos, and T. Palpanas. Indexing for interactive exploration of big data series. In *SIGMOD*, 2014.

[196] K. Zoumpatianos, S. Idreos, and T. Palpanas. Rinse: Interactive data series exploration. In *VLDB*, 2015.

[197] K. Zoumpatianos, S. Idreos, and T. Palpanas. Ads: the adaptive data series index. *The VLDB Journal (accepted for publication)*, 2016.

[198] K. Zoumpatianos, Y. Lou, I. Ileana, T. Palpanas, and J. Gehrke. Generating data series query workloads. *Under Submission*, 2016.

[199] K. Zoumpatianos, Y. Lou, T. Palpanas, and J. Gehrke. Query workloads for data series indexes. In *KDD*, 2015.

[200] K. Zoumpatianos, T. Palpanas, and J. Mylopoulos. Strategic management for real-time business intelligence. In *Enabling Real-Time Business Intelligence Workshop (BIRTE), held in conjunction with VLDB*, 2012.

[201] K. Zoumpatianos, T. Palpanas, J. Mylopoulos, A. Maté, and J. Trujillo. Monitoring and diagnosing indicators for business analytics. In *CASCON*, 2013.